Doctoral Thesis

# Efficient Distributed DNN Training through Resource-Aware Hybrid Parallelism

Jay H. Park

Department of Computer Science and Engineering

Ulsan National Institute of Science and Technology

2021

# Efficient Distributed DNN Training through Resource-Aware Hybrid Parallelism

Jay H. Park

Department of Computer Science and Engineering

Ulsan National Institute of Science and Technology

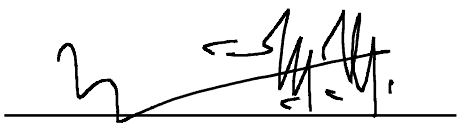
# Efficient Distributed DNN Training through Resource-Aware Hybrid Parallelism

A dissertationsubmitted to
Ulsan National Institute of Science and Technology
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Jay H. Park

07.07.2021 of submission

Approved by



Advisor

Sam H. Noh

# Efficient Distributed DNN Training through Resource-Aware Hybrid Parallelism

Jay H. Park

This certifies that the dissertation of Jay H. Park is approved.

07.07.2021 of submission

Signature

Advisor: Sam H. Noh

Signature

Young-ri Choi

Signature

Woongki Baek

Signature

Myeongjae Jeon

Signature

Jiwon Seo

# Abstract

Deep Neural Networks (DNN) have been popularly used to solve various problems such as image classification, speech recognition, topic modeling, and text processing. The size of DNN models models have continuously been growing in size in order to improve the accuracy and quality of models and to deal with complex features of data. The size of input dataset have also increased to achieve higher accuracy. Training using a large DNN model requires a significant amount of dataset and computation. This is generally achieved through distributed parallel execution, and various traditional distributed parallelization methods have been proposed. Distributed parallelism method should be appropriately applied according to the characteristics of the DNN and computing resources. The goal of this dissertation is to propose a hybrid parallelization suitable for the characteristics of DNN and the condition of computing resources. Also, to train, a large amount of input dataset should be pre-processed, which also takes a lot of time. It describes the parallelization approaches from dataset pre-processing to the training process.

This dissertation presents three systems for efficient DNN training. First, we present a Resource-Aware Layer Placement (RALP) scheme to alleviate the network bottleneck and balance computation to improve performance for CNN distributed training. Partial data parallelism can be applied through automatic layer placement of our RALP system. It shows more accelerated training performance for most CNN models than when using traditional data parallelism. Second, we present HetPipe, a novel system for large DNN training in heterogeneous GPU clusters. It integrates pipelined model parallelism and data parallelism to train large DNNs by effectively utilizing the resources of heterogeneous GPUs. We also propose a novel parameter synchronization model, referred to as Wave Synchronous Parallel (WSP), to accommodate pipelined model parallelism and data parallelism. Finally, we present a resource-adaptive parallelization approach for pre-processing datasets. Parallelization was made possible by analyzing the pre-processing of the DNN-based recommendation system, which had not previously been able to apply parallelization. Furthermore, we propose a Resource-Adaptive Pre-processing (RAP) system that monitors the CPU, memory, and storage computing resources and can automatically adjust the parallel degree for pre-processing datasets.

# Dedication

I dedicate this dissertation to my parents.
For their unconditional love, support, and encouragement.

제 박사학위 논문을 부모님에게 바칩니다.

# Contents

# List of Figures

# List of Tables

# I Introduction

Deep Neural Networks (DNN) have been popularly used to solve various problems such as image classification [1, 2], speech recognition [3], topic modeling [4], and text processing [5]. The size of DNN models (i.e., the number of parameters) have continuously been increasing in order to improve the accuracy and quality of models and to deal with complex features of data [6–9]. The size of input dataset and batches used for training have also increased to achieve higher accuracy and throughput [6, 10]. To achieve high prediction accuracy through training, large scale machine learning models such as deep learning are used [11]. Training using a large scale model requires a significant amount of dataset and computation. This is generally achieved through distributed execution [11–14] or by making use of GPUs [15–17]. For such large scale executions, machine learning frameworks such as TensorFlow [18], PyTorch [19], and Caffe [20] are openly available.

There are various parallel modes for distributed training. For training large DNN models, Data Parallelism (DP) [13, 21–23], which employs multiple workers using parameter servers or AllReduce communication, and Model Parallelism (MP) [12, 24, 25], which divides the network layers of a DNN model into multiple partitions and assigns each partition to a different GPU, have commonly been leveraged. Furthermore, to mitigate the critical issue of low GPU utilization of naive model parallelism, Pipelined Model Parallelism (PMP), where minibatches are continuously fed to the GPUs one after the other and processed in a pipelined manner, has recently been proposed [6, 26].

This dissertation tries to understand the problems caused by using an increasingly larger input dataset and a larger DNN model and solve it with hybrid parallelism. Hybrid parallelism is applied from the preprocessing of the input dataset to the DNN training stage. As there are several parallelism modes, it is crucial to apply a parallelism method suitable for the DNN condition. The problem is solved by applying resource-aware hybrid parallelism to appropriate parallelism according to the computing resources and the characteristics of the DNN model.

Depending on the DNN model and resource condition, we apply appropriate hybrid parallelism. In this dissertation, we will describe three works. 1) In the case of the Convolutional Neural Network (CNN) model, we propose a partial DP system that applies DP to particular layers of the CNN in consideration of network resources. 2) In the case of a large DNN model, we present a system that integrates PMP and DP considering heterogeneous GPU resources. 3) Finally, resource-adaptive parallelism was developed as a system for pre-processing a large number of datasets. It automatically adjusts the degree of parallelism according to CPU, memory, and storage resources to maximize resource utilization. The following sections describe the contributions to each work.

This dissertation integrates and updates upon the work previously published at [27, 28].

## 1.1 Contributions

This dissertation presents new systems for *efficient distributed DNN training through resource-aware hybrid parallelism*. The contribution of this dissertation are as follows.

**Accelerate CNN distributed training by reducing network traffic through partial DP.** In this dissertation, first, Convolutional Neural Network (CNN) model analysis, we find that placement of layers can have a considerable effect on performance. We propose RALP (Resource-Aware Layer Placement), a scheme to reduce network traffic through layer placement that considers the resources that each layer uses. RALP applies partial data parallelism to reduce network traffic. It provides an automatic layer placement that allows partial DP, which shows that it is more effective in using network resources than traditional DP. Finally, we incorporate RALP within the TensorFlow framework such that layers can be automatically placed for more efficient training. Our evaluation with RALP shows that training time can be significantly reduced without loss of accuracy for many CNN models.

**Enabling large DNN training on heterogeneous GPU clusters through integration of PMP and DP.** In this dissertation, we investigate how to enable training of large DNN models on a heterogeneous GPU cluster that possibly includes whimpy GPUs that, as a standalone, could not be used for training. We present a DNN training system, *HetPipe* (*Het*erogeneous *Pipe*line), that integrates PMP with DP. In HetPipe, a group of multiple GPUs, called a *virtual worker*, processes minibatches in a pipelined manner, and multiple such virtual workers employ data parallelism for higher performance. We also propose a novel parameter synchronization model, which we refer to as Wave Synchronous Parallel (WSP) to accommodate both PMP and DP for virtual workers. Our experimental results on a given heterogeneous setting show that with HetPipe, DNN models converge up to 49% faster compared to the state-of-the-art DP technique.

**Resource-adaptive parallelism for efficient dataset pre-processing.** We analyze dataset pre-processing of DNN-based recommendation system with three insights. Through this analysis, the existing serial execution can be parallelized. Furthermore, we propose RAP (Resource-Adaptive Pre-processing) to determine the degree of parallelism by considering the resource state of CPU, memory, and disk. RAP monitors resource status in real-time and performs parallel pre-processing suitable for resource status by adjusting the degree of parallelism. In addition, distributed pre-processing in multi-node is implemented, and the experimental results are shown. Our experimental results show that by applying four distributed pre-processing, the pre-processing that took 112 hours for serial execution was reduced by up to 97% to 4 hours.

## 1.2 Organization

This dissertation is organized as follows. In the next section, we briefly discuss background of DNN necessary to understand this study. In Section III we discuss the detailed analysis of CNN, and present the design of *Resource-Aware Layer Placement* (RALP). Section IV present a large DNN training system, *HetPipe*, that integrates PMP with DP. Regarding dataset pre-processing, in Section V analyzes dataset pre-processing of DNN-based recommendation system. Based on this analysis, we present *Resource-Adaptive Pre-processing* (RAP) with resource-adaptive parallelization applied. Previous studies related to our work are summarized in Section VI, and we conclude with Section VII.

# II Background

## 2.1 DNN Training

Before training the DNN model, we need to pre-process the dataset. Pre-processing is used when converting datasets into binary forms to improve training speed or converting raw datasets into input forms necessary for training specific DNN models. Training proceeds after pre-processing the raw dataset. The goal of training of a DNN model composed of multiple layers is to find the parameters (or weights) $w$ of the model that minimizes the sum of a loss function for the training dataset that consists of training samples and their labels. In a popularly used training method, *stochastic gradient descent* (SGD), it computes the weight updates, i.e., *gradients* on a subset of training samples, called a *minibatch*, and updates weights $w$.

The training process consists of a *forward pass* and then a *backward pass*. In the forward pass, the model first predicts the label for each of the samples in a minibatch. Each layer computes activations for the next layer using the given input data and the current parameters. Finally, the last layer of the model computes loss based on the predicted and actual labels. In the backward pass, the loss is backpropagated over all the layers of the model where each layer computes gradients using the gradients computed by the upper layer and activations previously computed in the forward pass.

## 2.2 Distributed Parallel Training for DNN

As DNN models become more sophisticated and are trained on larger datasets, it is becoming common to scale training across machines in a distributed setting. This section explains a general parallelization approach for distributed training.

**Data Parallelism.** Data parallelism (DP) utilizes multiple workers to speed up training of a DNN model. It divides the training dataset into subsets and assigns each worker a different subset. Each worker has a replica of the DNN model and processes each minibatch in the subset, thereby computing the weight updates. Therefore, if a DNN model cannot be loaded into the memory of a single GPU, DP cannot be used.

Among the multiple workers, the parameters are synchronized using parameter servers [13] or AllReduce communications [22, 23]. For *Bulk Synchronous Parallel* (BSP) [18, 29], each worker must wait for all other workers to finish the current minibatch $p$ before it starts to process the next minibatch $p+1$ so that it can use an updated version of the weights for minibatch $p+1$. For *Asynchronous Parallel* (ASP) [18, 30], each worker need not wait for other workers to finish minibatch $p$, possibly using a stale version of the weights. With BSP, which is possible for both the parameter servers and AllReduce communications, the system may suffer from high synchronization overhead, especially in a heterogeneous GPU cluster where each worker with a different GPU provides different training performance [31]. On the other hand, while ASP, which is possible for the parameter servers, has no synchronization overhead, it is known that ASP does not ensure convergence [30, 32].

A method that takes the middle ground between BSP and ASP is *Stale Synchronous Parallel* (SSP) [33]. With SSP, each worker is allowed to proceed the training of minibatches using a *stale* version of the weights that may not reflect the most recent updates computed by other workers. Thus, workers need not synchronize with other workers whenever it finishes the processing of a minibatch. As such, parameter staleness can occur. However, this staleness is bounded as defined by the user and referred to as the *staleness threshold*. As SSP is beneficial when worker performance is varied, it has been explored especially in the context of heterogeneous systems [34].

In SSP, each worker periodically pushes the weight updates to the parameter server. This synchronization interval is called a *clock*. Thus, each worker increases its local clock by one for every iteration, which is the training period of a minibatch. For a given staleness threshold $s$ where $s \geq 0$, each worker with clock $c$ is allowed to use a stale version of the weights, which includes all the updates from iteration 0 to $c-s-1$ and, possibly, more recent updates past iteration $c-s-1$. That is, a worker can continue training of the next minibatch with parameters whose updates may be missing from up to the $s$ most recent minibatches.

**Model Parallelism.** Model parallelism (MP) is typically exploited for large DNN models that are too large to be loaded into memory of a single GPU. In particular, a DNN model composed of multiple layers is divided into $k$ partitions and each partition is assigned to a different GPU. Each GPU executes both the forward and backward passes for the layers of the assigned partition. *Note that it is important to execute the forward and backward passes of a partition on the same GPU* as the activation result computed for the minibatch during the forward pass needs to be kept in the GPU memory until the backward pass of the same minibatch for efficient convergence, as similarly discussed by Narayanan and others [26]. Otherwise, considerable extra overhead will incur for managing the activation through either recomputation or memory management.

In the basic form of MP, $k$ GPUs, individually, act as one *virtual worker* to process a minibatch as follows: For each minibatch, execution of the forward pass starts from $\text{GPU}_1$ up to $\text{GPU}_k$. When each $\text{GPU}_i$, where $1 \leq i < k$, completes the forward pass of the assigned partition, it sends the computed activations of *only the last layer in its partition* to $\text{GPU}_{i+1}$. Once $\text{GPU}_k$ finishes the forward pass of its partition, the backward pass of the minibatch is executed from $\text{GPU}_k$ down to $\text{GPU}_1$. When each $\text{GPU}_{i'}$, where $1 < i' \leq k$, finishes the backward pass, it sends the computed local gradients of *only the first layer in its assigned partition* to $\text{GPU}_{i'-1}$. This basic form of MP results in low GPU utilization as only one GPU is actively executing either the forward or backward pass. Nonetheless, MP allows execution of large DNN models that are too large for a single GPU.

**Pipelined Model Parallelism.** To improve utilization of the GPUs in a virtual worker, minibatches can be processed in a pipelined manner. The subsequent minibatches are fed into the first GPU in MP (i.e., $\text{GPU}_1$) one by one once the GPU completes the processing of the previous minibatch. This allows for multiple GPUs to simultaneously execute either the forward or backward pass of their assigned layers for different minibatches. This is referred to as Pipelined Model Parallelism (PMP).

# III RALP: Accelerated Training for CNN Distributed Deep Learning

In recent years, machine learning has been used to solve problems in many fields [35–39]. In particular, Convolutional Neural Networks (CNN) has been popular in vision and audio recognition areas [1,2,40]. Various machine learning models go through a training phase, which makes use of training data, that requires many iterative computations through the model. This is generally achieved through distributed execution. Many modern distributed machine learning systems make use of the parameter server architecture [12–14, 16, 17, 41]. With this architecture, data parallelism and model parallelism is used to compute a significant amount of training data [12]. For data parallelism, the model itself is replicated, while for model parallelism, the model is partitioned on multiple GPUs and run in parallel. Model parallelism can be useful for large models. However, it is rarely used as the performance gains have been shown to be minimal due to heavy network traffic [42]. In this study, we focus on data parallelism in a distributed parameter server architecture.

Training in such an architecture, however, has its own limitations. The first is network traffic. The network can become the bottleneck as the workers must communicate and synchronize with the parameter server at each training step. Such network traffic tend to grow with the number of workers in case multiple parameter servers are deployed for large parameter sizes [43]. The second limitation comes with the issue of placement. That is, models are placed at particular nodes such as servers with GPUs and what not. Such placement decisions affect computation time as well as network communication as these machines may need to transfer data among themselves. Thus, ineffective placement can lead to heavy communication, leading to degraded performance. Such placement decisions can be difficult to make if one is not familiar with the characteristics of the model [44].

The goal of this study is to improve performance in distributed deep learning training by alleviating the network bottleneck and balancing computation, with a focus on CNN. We propose a Resource-Aware Layer Placement (RALP) scheme that minimizes network traffic, while considering the computation overhead incurred by the placement. To develop this scheme, we go through a careful analysis of various CNN models, where we characterize the computational and communication needs of the layers that comprise CNN. We find common traits that may be exploited in placing the layers. In particular, we find the fully connected layer is the best fit to be placed in the parameter server. We implement RALP in the TensorFlow framework such that the nodes comprising particular layers may be placed in appropriate servers for improved performance. Our experimental results show that, while not all models can benefit from RALP, many models are able to reap significant performance improvements, improving by as much as 13$\times$ compared to the baseline TensorFlow framework.

The rest of Section III is organized as follows. In the next section, we discuss CNN as well as distributed deep learning training as background. In Section 3.2, we discuss the motivation behind our study, focusing on detailed analysis of various components of CNN that affect performance. Based on this analysis, we present the design of RALP in Section 3.3. In Section 3.4, we present experimental results and finally, in Section 3.5, we conclude with a summary.

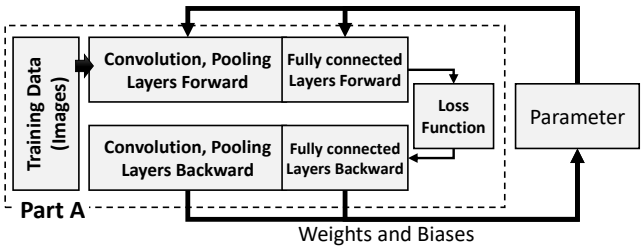



Figure 1: CNN model training structure

### 3.1 Training Convolutional Neural Networks

**Convolutional Neural Networks.** Convolutional Neural Networks (CNN) is a class of neural network used in tasks ranging from image classification [1, 2, 39, 40, 45] to video recognition [46, 47] to recommender systems [48]. Figure 1 shows the workflow of conventional CNN model training and the various stages that it goes through during execution.

With the training data (e.g., images) as input, model training proceeds through a forward pass and a backward pass in an iterative fashion. First, the feature of the input data is extracted through multiple layers in the *forward pass* of model training. Using the extracted feature, a loss function calculates the loss value, which is a similarity measure between the predicted value and the actual output value of classification. Then, the *backward pass*, also referred to as *back propagation*, computes a model update by going through the forward pass layers in reverse order. During the backward pass, each layer obtains the gradient of the parameters (i.e., weights and biases) and uses optimization methods such as stochastic gradient descent to minimize the loss value. This process is repeated until the convergence of training loss.

Figure 1 shows that the forward-backward computation in CNN model training largely consists of convolution layers, pooling layers, and fully connected layers. The convolution layer processes the convolution operation that extracts high-level features of the input data. The pooling layer is where the given space is reduced in the horizontal and vertical directions by taking the maximum or the average of values in the target space through max or average operations. The pooling layer commonly comes after the convolution layer. The fully connected layer connects neurons of adjacent layers and can arbitrarily set output neurons to connect for each input neuron. As all the neurons are connected, they contain a large number of parameters. The last layer of a CNN model is generally a fully connected layer that outputs the predicted value.

**Distributed CNN Training.** As deep learning models become more sophisticated and are trained on larger datasets, it is becoming common to scale training across machines in a distributed setting. The most common form of such Distributed Deep Learning (DDL) training is *data parallelism* [49] where multiple workers train on their own set of data in parallel. Essentially, each worker loads a complete copy of the model (Part A in Figure 1) into its own memory of an accelerator such as a GPU that is assigned to the worker. In each training iteration (or step), each worker performs training using a subset of the

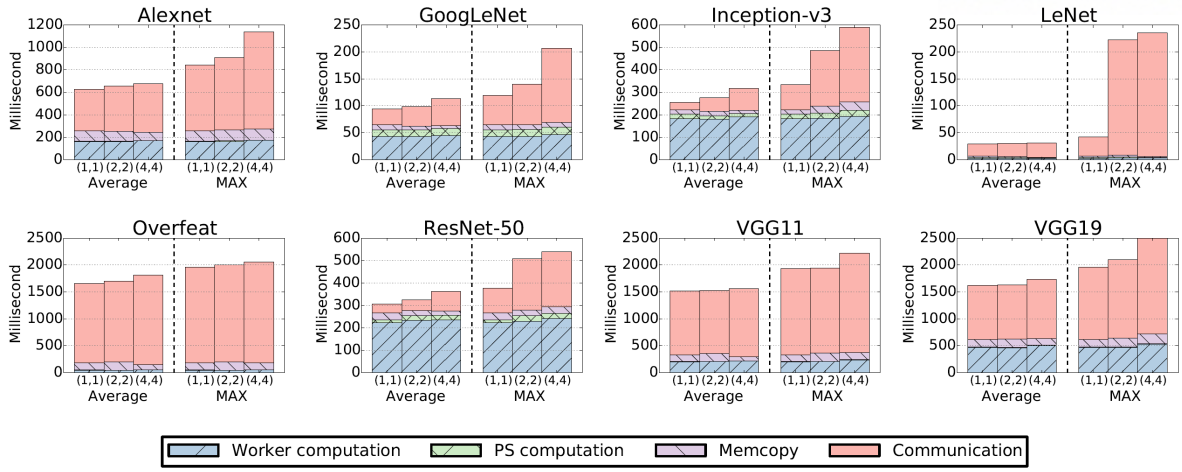



Figure 2: Distributed training: time breakdown (after 100 step executions) including network transfer time, where the left bars are average step execution time for all workers while right bars are the average step execution time for the slowest worker. Numbers in parentheses represent number of workers and PSes.

input data that is equally divided among all workers. At the end of the iteration the workers exchange gradients to synchronize (or *aggregate*) model updates.

A typical form of model aggregation is the parameter server (PS) architecture [13], which is popular in production systems [49, 50]. In this architecture, the PS hosts the master copy of the DDL model and is in charge of updating the model using the local results sent from all workers. The workers pull back the updated model from the PS at the beginning of each iteration and proceeds through the next iteration. Note that a single DDL training job can use multiple PSes [43, 49].

### 3.2 CNN Characterization

In this section we discuss the characteristics of training for CNN distributed deep learning (DDL) that stands as motivation behind the design of RALP. We focus our discussion on communication between workers and PSes during model aggregation and the computation involved at each worker. For this study, we use the CNN model benchmarks [51] supported in TensorFlow 1.12 and the ImageNet-1K dataset [52]. We perform experiments on a cluster of 8 machines connected over a 56 Gbps InfiniBand network, where each machine is equipped with 4 NVIDIA TITAN Xp GPUs. This is the same experimental setup used in Section 3.4.

**High Communication Costs in CNN DDL Training.** In CNN DDL training, the model aggregation phase is part of the critical path, having a significant hindering effect on training progress. To illustrate, consider a job using data parallelism where all workers must communicate with the PS to synchronize all the parameters of the model in each training step. After each worker calculates the gradients and updates its model, it sends gradients to the PS. Once the parameters are aggregated, these values are sent back to the individual workers. That is, communication between workers and the PS occurs twice

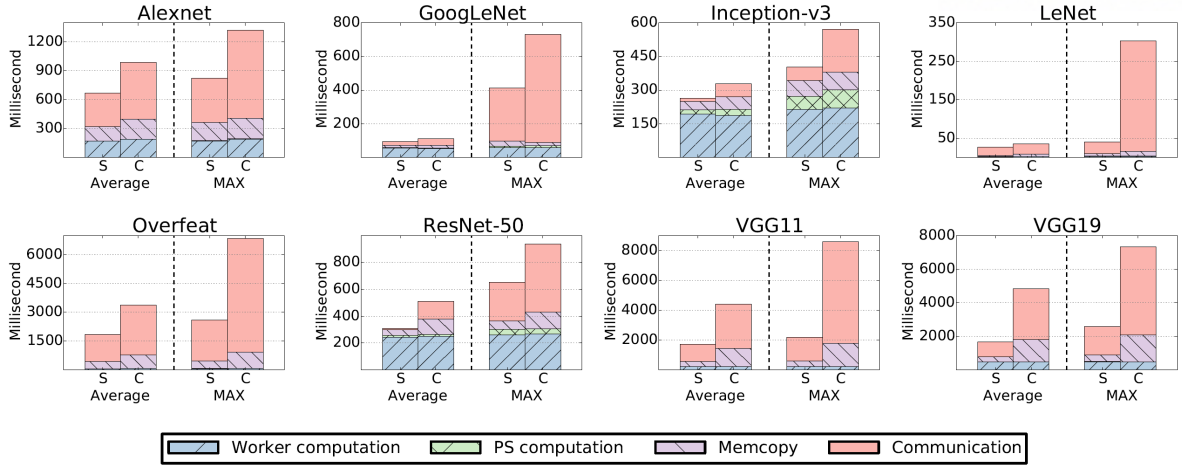


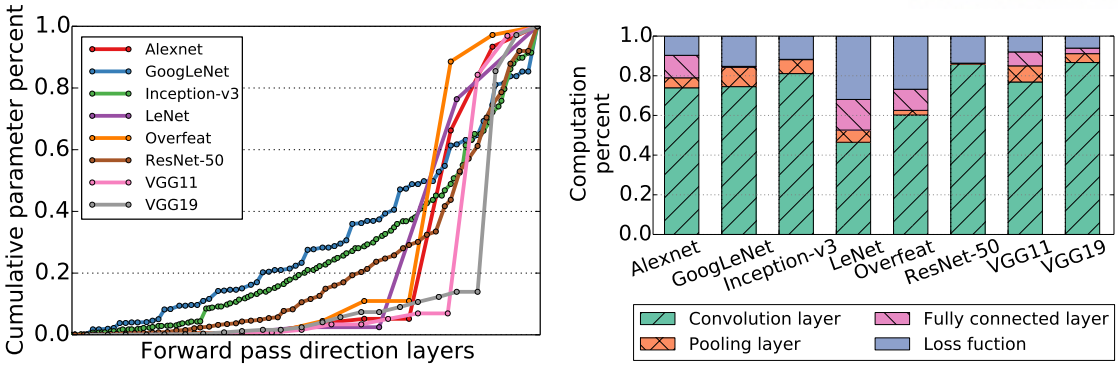

Figure 3: Network interference: S represents single model training, C represents consolidated training with 8 models in a cluster.

at every training step, and the amount of network transfer is proportional to the number of workers.

To assess this overhead, we perform a set of experiments, whose results are shown in Figure 2, where we break down the end-to-end time spent for each worker to finish a training step, including the time spent for model aggregation. More specifically, the elapsed time is divided into four categories according to the type of operation performed: (i) **worker computation** for model training by the worker; (ii) **PS computation** for the aggregation step in the PS; (iii) **memcopy** for copying data between the host machine and the GPU; and (iv) **communication** for transferring model parameters over the network. The results for the first three operations are provided by the TensorFlow timeline tracer, while those of the last operation are obtained by measuring the end-to-end time for training and then subtracting the elapsed times of the first three operations.

We train 100 steps for each worker and report, on the left bars, the average of each step over all workers, and, on the right bars, the average (over the 100 steps) times of the slowest worker, which essentially determines the performance when all workers need to synchronize training progress at each step [53]. Our measurements are based on changing the number of workers, denoted by the left number in the parenthesis, with the number of PSes (right number) to be the same as the number of workers as guided by prior work [43]. The workers and PSes are all spread out on different machines.

Figure 2 shows that network communication performed in model aggregation dominates for a number of scenarios across the benchmarks. This happens even when a single worker and PS is used, which ought to produce the least amount of network traffic. In particular, we observe that the average fraction of time used for communication is 53% across the benchmarks. As model training scales out, we see an increase in both the absolute amount of time required for training and the dominance of communication time. For example, when using 4 workers and 4 PSes, communication makes up, on average, 58% of the total training time per step, and it is worse for the slowest worker (in MAX), with LeNet (4,4) taking up as much as 92% of the total time. Our observation here is that in order to deliver high performance distributed training, it is crucial to reduce the volume of network transfer.




(a) Communication parameter

(b) Computation time

Figure 4: Memory usage and computation of CNN

**Network Interference in Shared Servers.** In a cluster of machines, consolidation of different jobs on the same cluster could lead to interference due to contention of shared network such as RDMA [49, 50, 54]. Such sharing could further deteriorate training performance that is already hampered by network communication overhead. To confirm that workload consolidation indeed relates to further network interference and consequently, training progress, we make another set of measurements similar to Figure 2, but this time on 3 worker, 1 PS training. Figure 3 compares the average execution time of 8 jobs that run concurrently on our 32-GPU cluster (denoted by C) to that of a job that runs in isolation (denoted by S).

The results in Figure 3 confirm that jobs further interfere with each other in the network and that interference among jobs on shared resources considerably slows down training time. For example, in our experiments, LeNet slows down by over 7×. We also see that this inefficiency mostly comes from the time prolonged during model aggregation due to increased network contention. Thus, we conclude that optimizing model aggregation could improve performance not only when training runs in isolation, but also when workload is consolidated to share network resources.

**Disproportional Memory Usage and Computation.** Another key characteristic of CNN relates to how parameters and computation are laid out among the layers that comprise CNN. We now elaborate on these points.

First, for a number of CNN models, a significant fraction of memory is used by a few layers exercised in the last phase of the training's forward pass. Figure 4(a) shows the cumulative distribution of memory usage in terms of parameter sizes in the order of the forward pass layers for each of the models. Overall, the figure shows that a majority of CNN models have skewness in memory usage, with the last 25% of the layers occupying more than 86% of memory space in 5 out of 8 models. On inspecting these memory-demand layers by layer type, we find that they correspond to the fully connected layers. Thus, there is an opportunity to reduce network transfer during model aggregation by placing the fully connected layers within the PS machine.

Let us now consider the computation distribution in CNN. Figure 4(b) shows the breakdown of computation time in CNN training by layer type. We see that a large fraction of the computation time is

occupied by the convolution layers. Specifically, the convolution layers account for around 80% of the total GPU time used during training on average. This characteristic makes colocating the fully connected layers with the PS a more promising approach, because those computation-demand convolution layers mainly appear in the first phase of training's forward pass and do not interfere with computation on the fully connected layers. Note that it may be possible to accelerate the execution of the convolution layers by assigning more compute resources (e.g., GPUs) to these layers separately through layer partitioning.

**Challenges.** In this section, we have shown that network communication is a major performance factor in CNN training no matter the model is trained in isolation or in consolidation with others. We also showed that a large proportion of the parameters, which need to be communicated to the PS, are concentrated in the latter layers. This provides an opportunity to reduce network overhead by placing these layers within the PS's machine. Also, we showed that the convolution layers are concentrated in the front phase of training, and thus, again, there is an opportunity to optimize computation.

However, blindly offloading the latter layers to the PS machine may not pay off for all CNN models as some of these models do not entirely correspond to the structure shown in Figure 1. For example, for some CNN models, the fully-convolutional network replaces the fully-connected layers with convolutions to take in an arbitrary input size for efficient inference and learning [55]. Also, in the ResNet model, the network has many filters in the convolutional layer, and hence, many parameters, which requires sizable memory space [1]. In these cases, either there is no significant skewness in parameter distributions among layers or the earlier part in the CNN pipeline could consume considerable memory, thus affecting the synchronization cost. The main goal of our work is to provide a systematic way to identify models that could benefit from RALP and structure a pipeline of layers across distributed resources to obtain higher training performance.

### 3.3 Layer Placement with RALP

We propose RALP — Resource-Aware Layer Placement — a scheme whose target is to reduce network communication and balance computation to expedite distributed training of CNN. To achieve the goal, RALP carefully exploits workload characteristics studied in Section 3.2. RALP places CNN layers, which is composed of a communication and computation sequence, in particular machines in a way that data flows efficiently and compute-bound layers are executed in parallel. Typically, a deep learning model is composed of a number of layers, and it is feasible to place each individual layer anywhere among compute resources such as GPUs. However, since our focus is on CNN, we actively utilize CNN-specific properties related to the model structure (shown in Figure 1) in the optimization. Nonetheless, we will briefly discuss how to apply RALP beyond CNN.

Since network communication is on the critical path of training in distributed mode and memory-demand layers are the root cause, RALP takes an approach that colocates those layers with the parameter server (PS) in the same machine. In this way, the cost associated with model aggregation can be considerably reduced. Further, RALP decides for a given model if the model could benefit sufficiently from

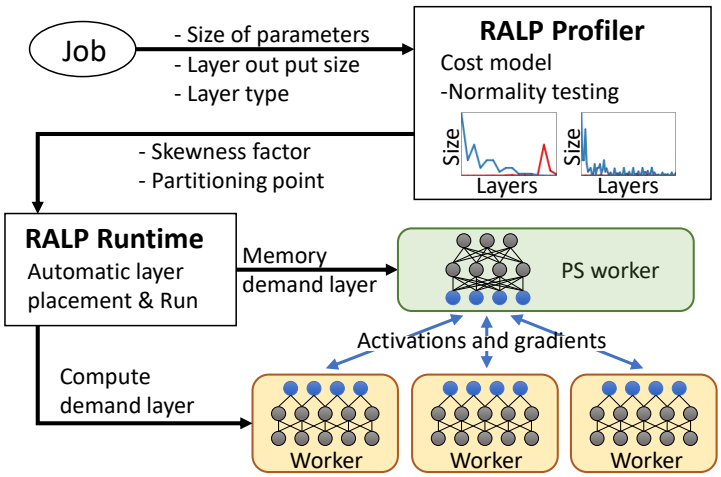



Figure 5: Workflow of resource-aware layer placement for distributed training in RALP

our resource-aware placement, and if so which "specific" layers it needs to offload to the PS machine to have most benefit.

When deciding layer placement, RALP also factors in such layers that are computationally expensive. To illustrate, consider a naive approach that places all layers with the PS. Evidently, this is not a good solution as this will incur considerable computational overhead on the PS machine. The natural choice for layer placement would be to place the layers with small computation, but with large memory demands (i.e., parameters) in the PS machine. This will have the effect of reducing network communication without overloading the machine that runs both PS and the offloaded layers.

### 3.3.1 System Overview

Figure 5 depicts the overall architecture of RALP and its two key components, RALP Runtime and RALP Profiler.

(i) **RALP Runtime** generates a distributed training plan and executes it on distributed compute resources. Training in RALP differs in several significant ways from the traditional PS-based approach. Previously, as the number of workers increases, it is common practice to use the same number of PSes [43, 49]. This is partly because network transfer is the bottleneck and using few PSes could make the PSes overwhelmed by the large volume of data to receive, process, and send back. In RALP, we can initiate distributed training efficiently using a smaller number of PSes for two reasons: (i) network transfer is no longer the bottleneck, and (ii) the PS machine that includes the memory-intensive layers (e.g., the fully connected layers) is not compute heavy [24, 56]. In effect, RALP can save resources for PSes while using the same number of workers, which mostly run the compute-intensive layers (e.g., the convolution layers).

(ii) **RALP Profiler** profiles the input CNN to estimate network communication across layers, and then informs where to *partition* between the layers to be network-cost effective. Blindly applying RALP to every CNN model will deteriorate some models that do not result in a sizable reduction in data transfer. Thus, our profiler uses a cost model that informs if there exists such gain through partitioning.

Table 1: Parameter skewness for 8 benchmarks under our study: larger absolute values indicate higher skewness.

| | Alexnet | GoogLeNet | Inception-v3 | LeNet | Overfeat | ResNet-50 | VGG11 | VGG19 |
|---|---|---|---|---|---|---|---|---|
| Skewness factor | −2.27 | −0.74 | −0.96 | −1.16 | −2.11 | −1.26 | −3.62 | −3.02 |

Essentially, the cost model exploits the parameter size for each layer and additionally the size of activations and gradients exchanged between two neighboring layers during the forward-backward pass. In addition, the profiler takes computation costs into account in a simple yet effective way based on CNN's characteristics: we avoid partitioning layers when the optimal point for splitting the model is found between the convolution layers, which are computationally costly. This makes the profiler widely applicable as it depends only on the model itself including training configuration (e.g., batch size), not on dynamic factors such as clock speed of in-use GPUs.

**Workflow.** Figure 5 shows the stages that RALP goes through. Once a job is received, it is delivered to RALP Profiler, which determines if partitioning is adequate or not. Jobs can bypass the profiler if the characteristics of the jobs are already known, or has been estimated offline, to benefit from partitioning. The profiler consults the cost model to determine the eligibility of applying RALP. Only if the job is predicted eligible, the model is partitioned and placed across particular machines; otherwise, no partitioning occurs, and the model training runs on the standard PS architecture.

In the subsequent sections, we explain distributed training performed in RALP in more detail.

### 3.3.2 Distributed Training

Figure 5 also shows the training procedure and communication of distributed training enabled by RALP Runtime. For simplicity, our explanation is based on one PS and multiple workers, a default configuration on a moderate degree of parallelism (e.g., 32 workers). We further discuss how to enable multiple PSes in RALP in case training must scale on much higher degree. We henceforth use the term *PS worker* to explicitly denote the training procedure ongoing in the PS machine.

**Training Procedure.** Workers run computation-demand layers and require enough compute resources to speed up the training progress. We follow conventional data parallelism to run workers concurrently. Workers must communicate with the PS worker independently during ongoing forward-pass and backward-pass computations.

In each training step, the PS worker receives activations from all workers, which arrive at the PS worker in an arbitrary order. In order to make sure that the backward pass computes the gradients using the same version of weights accessed in the forward pass, the processing of incoming activations is not interleaved, i.e., one batch of activations from a worker is processed at a time. As the memory-demand fully-connected layers in the PS machine are not computation-intensive [56], this processing is typically fast and does not add back pressure.

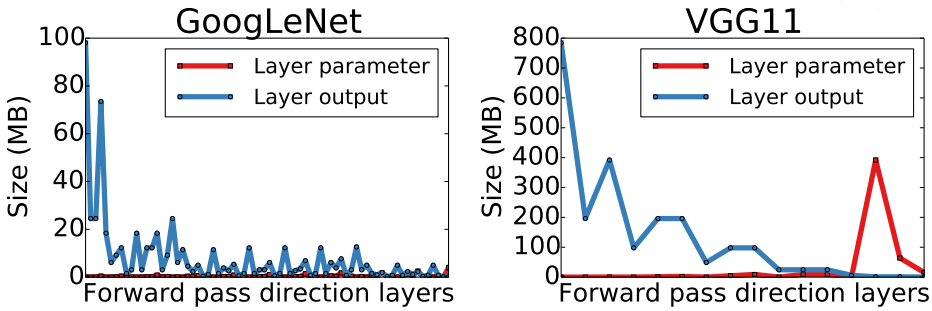



Figure 6: Distribution of parameters and output sizes across layers

**Model Aggregation.** We so far explained how workers communicate with the PS worker during training, and we now describe model aggregation occurring at the end of each training step. The most notable distinction from the original PS architecture is that aggregating the memory-demand layers is done through intra-machine data transfer. Workers may need to synchronize with the PS over the network, and in this case, the amount of parameters to transfer over the network is small. This synchronization is again fast, alleviating pressure on network usage.

**Enabling Massive Parallelism.** In training a single job on our 32-GPU testbed, we observe no significant performance issue with using a single PS worker while exploiting all other GPUs to run the workers of the job. *Thus, in the current evaluations, we primarily consider a single PS worker configuration.* Nonetheless, since the largest job reported in a production cluster runs up to 128 GPUs [49] and we expect larger scales in the future, we support RALP training on multiple PS workers to balance the load incoming from a large number of workers. To facilitate this, we partition computation previously done in the single PS worker so that the PS workers run in parallel while aggregating activations from workers.

Basically, RALP can use any form of model parallelism [6, 26, 57] to run memory-demand layers with high parallel efficiency. Among available options, we adopt pipeline parallelism since we observe that it leads to lower communication between the concurrent PS workers [58]. Note that in the use of $N$ PS workers, RALP will create $N$ parameter servers (PSes) to colocate each PS with a PS worker such that only intra-machine traffic is generated during model aggregation. That is, as RALP partitions the memory-demand layers among the PS workers, we partition their model parameters correspondingly across multiple PSes. As a result, irrespective of the number of PS workers, RALP is effective in reducing network usage, and consequently, improves training performance considerably as we discuss in detail in Section 3.4.

**Synergy with Cluster Resource Managers.** For distributed training, most deep learning frameworks require all GPUs be available at the same time [49, 50]. In traditional resource scheduling in a shared GPU cluster, one of the biggest concerns has been on job placement such that locality becomes as high as possible, i.e., packing the job's GPUs within as fewer machines as possible [49, 50]. This is because greater locality improves training time by bringing down model synchronization overhead. However, in

practice, such locality constraints often need to be relaxed to reduce waiting times, especially for jobs that use many GPUs [50]. With RALP, the scheduler can facilitate relaxed locality without sacrificing much training performance as a training job can eliminate a significant fraction of network transfer.

### 3.3.3 Model Profiler

We now present a model profiler that decides resource-aware layer placement at runtime on each training job.

**Inputs.** Our model profiler takes three inputs to decide on the layer placement: (i) the size of parameters used in each layer, (ii) the network traffic each layer would invoke, that is, the output size of the layer, and (iii) the type of each layer. The output size of the layer is proportional to the batch size, which is the number of images used in one step of training. In CNN, the output size tends to decrease in the order of the forward pass layers as shown in Figure 6.

**Algorithm.** Upon training job arrival, RALP first obtains an ordered set of layers, $\{L_1, L_2 \cdots, L_N\}$, where $L_i$ is the $i$-th layer in the forward-pass layers. It then retrieves the parameter size $P_i$ and output size $O_i$ of each layer $L_i$ and identifies the layer type (e.g., convolution layer) associated with $L_i$. Using this information, RALP goes through the following steps.

(i) **Step 1:** It measures the layer parameter skewness $S$ of the CNN model to estimate what fraction of the model parameters are concentrated in the latter layers. Formally, we measure the skewness using normality testing (with degree 3) [59] represented as follows in our context:

$$\frac{\frac{1}{n}\sum_{i=1}^{n}\left(P_i-\bar{P}\right)^3}{\left(\frac{1}{n}\sum_{i=1}^{n}\left(P_i-\bar{P}\right)^2\right)^{3/2}} \tag{1}$$

where $\bar{P}$ is the mean of all $P_i$ covering all layers in the model.

The result of Equation 1 represents the skewness factor $S$ of the parameter size distribution across layers. As a general rule of thumb, if $S < 0$, the distribution is left-skewed (i.e., data concentrated on the latter layers), and the absolute value of $S$ decides how highly the distribution is skewed. Since CNN models under our study all show a large fraction of data used by the latter layers, as Table 1 reports, the $S$ values of all 8 benchmarks used in our study are below zero.

(ii) **Step 2:** RALP filters out some models that violate a predefined criteria. RALP steers a predefined threshold $K$ to decide how aggressively it enables layer placement. For this, RALP compares the threshold $K$ with the output of Step 1, $S$, and leverages a general rule of thumb that $-1 < S < -0.5$ indicates a moderate level of skewness [60]. Thus, by setting $K = -0.5$, RALP Profiler decides to parallelize models as long as they exhibit at least moderate skewness. In other instances, RALP Profiler could apply model partitioning very selectively to highly skewed models by setting $K$ to larger values.

(iii) **Step 3:** Next, RALP decides where to partition between the layers to be network-cost effective. This step requires RALP to scan all layers from $L_1$ to $L_N$, and estimate the network cost assuming that it

splits the model with respect to the encountered layer. RALP finds the layer that produces the smallest network cost in a single training step, calculated from the following equation:

$$\arg \min_{1 \le i \le N} \{O_i + \sum_{x=0}^{i} P_x\}$$

where $O_i$ is the output size of layer $L_i$ and $\sum_{x=0}^{i} P_x$ is the parameter size summed up to $L_i$ including the preceding layers, which may be transferred at model aggregation. Notice that the cost excludes constant factors such as the number of communication occurrences for a step. RALP reports the minimum-cost layer as a position to split as long as it is not between compute-demand layers.

### 3.3.4 Implementation

We implement RALP in the TensorFlow framework that organizes all the model layers and their communications as a dataflow graph. TensorFlow provides the user API `tf.device` as a means to place each layer of the dataflow graph to particular compute resources, e.g., particular GPUs. Ideally, this interface should suffice to implement RALP. However, layer partitioning using `tf.device` requires that users not only modify the code but also perform partitioning before the dataflow graph is actually generated. This introduces complications in implementing RALP with just `tf.device`, and thus we take a lower-level approach as explained below.

To enable RALP transparently to users, we make changes to `device_function()` in TensorFlow. This is a TensorFlow internal function that allows finer control at the node level, assigning nodes of the dataflow graph to devices for execution. Using `device_function()`, we place all nodes from the memory-demand layers which we determine through the skewness normality testing along with in the parameter server. In addition, we deal with two complementary modifications. First, we exclude some graph nodes that become useless if RALP is applied. Recall that the loss value is calculated in between the forward and backward passes. Some of these calculations make use of the compute-demand layer gradient values. Thus, nodes that compute the loss value based on the compute-demand layer gradient should not be placed on the PS worker. Second, there are extra nodes that need to be included in the worker's portion of the dataflow graph, though this applies only when there are multiple GPUs assigned to a worker within the same machine. In model aggregation, the average value of the gradients from these GPUs needs first to be calculated before transmitted to the parameter server. The nodes that perform these calculations are generated separately after the model graph is constructed, and put on the worker side only. Note that the PS worker does not need to include these nodes since RALP does not enable data parallelism for the PS worker.

Finally, to make use of RALP, we simply add a statement "`tf.name_scope('RALP'):`" to the model configuration code to specify the `name_scope`, then everything else is done automatically.

### 3.3.5 Summary

RALP improves conventional distributed training of CNN using three insights: (i) it colocates the memory-demand layers with PS in the same machine to significantly reduce network communication,

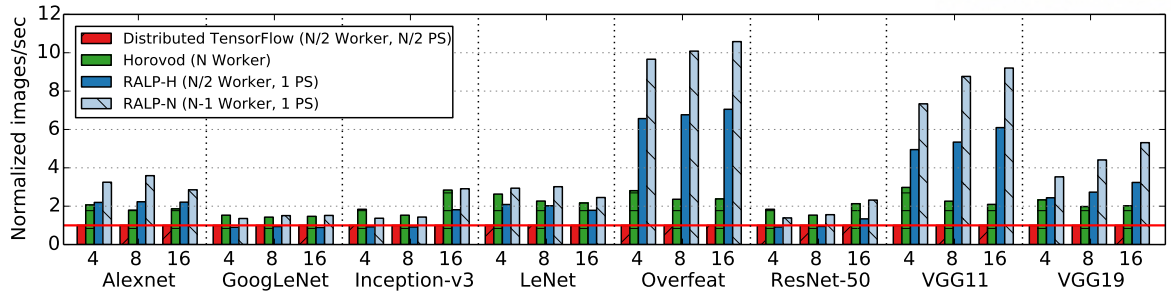



Figure 7: Training performance with ImageNet-1K for distributed TensorFlow (baseline), Horovod, and two configurations of RALP. The numbers on the $x$-axis represent the number of GPUs in use (denoted by N).

which used to be the main bottleneck; (ii) it assigns more resources to workers that run the compute-intensive layers while assigning fewer GPUs to PS workers, providing an efficient way of using available compute resources; and (iii) it profiles the CNN model using a well-defined cost model and guides partitioning when it turns out to be beneficial.

We believe that any machine learning framework can integrate RALP as long as it exposes the model internal structure. Moreover, our profiler and cost model are generally applicable to other classes of neural networks if network communication is prohibitively expensive. Our partitioning approach may need model-specific knowledge such as the computational cost of the layers to be offloaded; e.g., for CNN, we avoid offloading the convolution layers. Such information is easily obtainable from analyzing the given model.

## 3.4 Evaluation

In this section, we evaluate RALP on a number of benchmarks to show that it achieves higher throughput over the traditional PS architecture. We also show that RALP outperforms Horovod, which is managing communication bandwidth usage for DDL training. We then evaluate RALP for tasks that train on the dataset carrying out more complex classification, which we expect to be more prevalent in the future. Lastly, we evaluate the efficacy of RALP when multiple training jobs run simultaneously in a cluster sharing network, validating the significance of communication-aware layer placement on consolidated workloads.

### 3.4.1 Experimental Setup

**Testbed.** Each machine has two 2.10 GHz 4-core Intel Xeon processors and 64 GB of main memory, and 4 NVIDIA TITAN Xp GPUs each with 12 GB GPU memory. We conduct experiments on a cluster of 8 machines, thus 32 GPUs in total. The machines are connected via 56 Gbps InfiniBand network.

**Benchmarks and Datasets.** Each GPU in use runs either a worker and a parameter server (or PS). We vary the degree of data parallelism by changing the number of workers, and additionally the number of PSes.

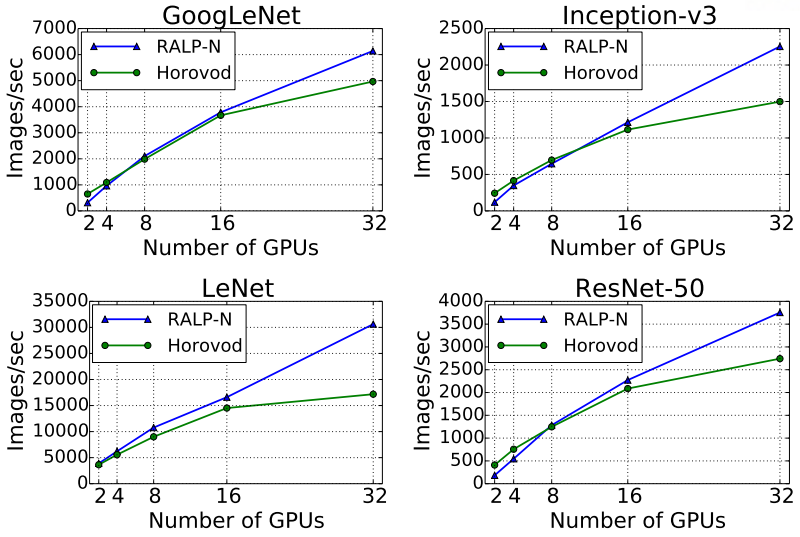



Figure 8: Performance comparison between RALP-N and Horovod while scaling up to 32 GPUs

Table 2: Transfer sizes of a 32-GPU training for one step.

| | Model Size | Horovod | RALP |
|---|---|---|---|
| Alexnet | 0.23 GB | 3.23 GB | 0.47 GB |
| Inception-v3 | 0.09 GB | 1.24 GB | 1.30 GB |
| VGG11 | 0.50 GB | 6.93 GB | 0.64 GB |

For evaluation we select the following 8 CNN benchmarks provided by TensorFlow 1.12 [51]: Alexnet [2], GoogLeNet [61], Inception-v3 [38], LeNet [62], Overfeat [63], ResNet-50 [1], VGG11 [40], and VGG19 [40]. We omit a few benchmarks that are variants of the selected benchmarks since their trends were very similar with the ones under the same umbrella: e.g., Inception-v3 and Inception-v4 exhibit similar performance changes with RALP. The batch size for each benchmark is set as the default value configured in TensorFlow 1.12. Unless specified otherwise, in RALP we configure the threshold value to compare with skewness factor as -0.5 to enable layer placement for all benchmarks (see Table 1).

By default we use the ImageNet-1K [52] dataset that classifies images into 1,000 categories. We use the 'ILSVRC 2017' [64] images as input data for training, where there are 1,281,166 images in total. For the experiments in Section 3.4.3, we use the ImageNet-22K [52] to evaluate RALP in the scenario where classification tasks become more complicated. Due to the output size that increases with the number of image categories, using ImageNet-22K needs to transfer more data over the network for model aggregation.

**Performance Metric.** As performance metric, we use throughput (images processed per second) across all workers at the training step. At the same time, we measure the data volume transferred between workers and PSes to measure the effect RALP has on mitigating network traffic. We collect these metrics averaged over 100 iterations after warming up the system through the first few iterations. We do not report convergence or accuracy of the trained model as classification accuracy remains the same.

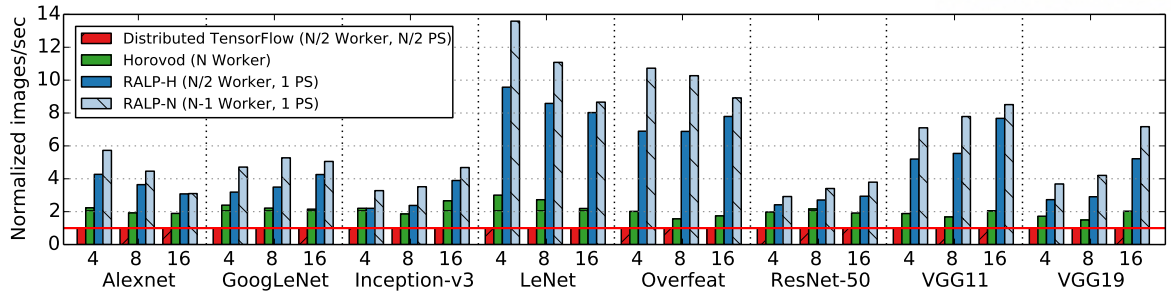



Figure 9: Training performance with ImageNet-22K for distributed TensorFlow (baseline), Horovod, and two configurations of RALP. The numbers on the $x$-axis represent the number of GPUs in use (denoted by N).

### 3.4.2 Single Model Training Performance

In this section, we look into the performance of RALP for isolated training on a variety of GPU assignment scenarios among workers and PSes. We carry out experiments by assigning GPUs for a training job on machines fully distributed across the network; we evaluated packing training job's GPUs onto a smaller number of machines and observed similar trends. The baseline for our comparison is distributed training that has been optimized to scale in the PS architecture in TensorFlow 1.12 [65]. Also, to maintain good training speed in distributed TensorFlow, given $N$ GPUs we assign workers and PSes an equal number of GPUs (i.e., $N/2$ GPUs) as was done in previous work [43, 49].

In RALP, in comparison, the efficient handling of network cost allows us to run fewer PSes. Hence, we evaluate RALP under two resource assignment scenarios: (i) $N/2$ GPUs for workers and 1 GPU for PS (i.e., total $N/2+1$ GPUs), which we denote as RALP-H (for half, though not exactly, of the GPUs), aiming at saving resources, and (ii) $N-1$ GPUs for workers and 1 GPU to PS (i.e., total $N$ GPUs), which we denote as RALP-N (for N GPUs), aiming at maximizing performance gain. We also show the performance of Horovod, accelerates distributed training through mitigation of network usage [22]. Horovod integrates a synchronization method called ring-reduce, which optimizes over all-reduce. As this method operates without parameter servers, for Horovod we use all the GPUs to run the workers only.

Figure 7 shows throughput results, normalized to the baseline, using 4, 8, and 16 GPUs for $N$. A few observations can be made from these results. First, we find that RALP-H and RALP-N both outperform the baseline for all cases, with models such as Overfeat and VGG11 showing speedup of 5 to 10 times. Thus, RALP can be used for either cost savings by using fewer GPUs (RALP-H) or more performance gains by using all assigned GPUs (RALP-N). For a few benchmarks the performance of RALP-H is almost identical to the baseline as the benefit of saving network through layer offloading in RALP is largely offset by the cost for transferring intermediate output data between workers and the PS. Nonetheless, we see that RALP-H can save resource usage up to 7 out of 16 GPUs, and the saved resources can be used for extra capabilities, e.g., improving capacity when consolidating training workloads.

Second, we see that the performance of RALP-N is almost always better than Horovod. For Alexnet, Overfeat, VGG11, and VGG19, the performance improvement is significant ranging from roughly 120% to around 958%. On the other hand, for GoogLeNet, Inception-v3, LeNet, and ResNet-50, performance improvements are marginal, and there are even points where Horovod performs better than RALP-N. However, once we increase the number of GPUs even further, we find that RALP-N starts to outperform Horovod by wider margins as shown in Figure 8. The primary factor that leads to such superior performance with RALP relates to how much communication is saved. Formally, Horovod transfers data as much as $2 \times S \times (W-1)$ for each training step, where S is the model size and W is the number of workers. Table 2 reports the calculated numbers for three benchmarks for Horovod when using 32 GPUs (i.e., W=32). Among the benchmarks, VGG11, which has the largest model size, has the biggest cost with 6.93 GB of data transferred at each step, which roughly translates to 8.97 GB/s of network transfer rate in our setup. In contrast, RALP's transfer volume is independent of the model size S. Through offloading layers to the PS machine, it avoids network transfer for large-size layers that decide most of the model size. Consequently, RALP could transfer much less data than Horovod as shown in Table 2. Our results are especially interesting as Horovod claims to scale well to the number of GPUs [22]. The results show that RALP scales even better.

Our final observation from Figure 7 is that RALP provides a setting for higher parallelism. As more GPUs are used to run more workers in RALP-N, we see a steady improvement in the relative throughput for most cases. This leads to the conclusion that managing network communication is crucial in distributed training as the degree of parallelism becomes higher. As a result, we can effectively steer RALP to trade off resource savings for higher performance when idle GPUs are ample.

**Can RALP decide not to partition models with either no or minimal benefits?** Yes. RALP can configure how aggressively it will perform layer placement by simply steering its threshold value to compare with skewness factor. For example, if the threshold is $-1.5$, RALP will trigger layer placement for Alexnet, Overfeat, VGG11, and VGG19 only, which have skewness factors smaller than $-1.5$. Notice that these benchmarks show substantial performance improvements as shown in Figure 7.

### 3.4.3 Increased Complexity in Classification

We now consider performance for a larger dataset. For this, we perform experiments with the ImageNet-22K dataset, which is a dataset with 21,841 classes using the 'ImageNet Fall 2011 release' image data. Since the TensorFlow CNN benchmark program does not provide standard models for ImageNet-22K, we modify the number of classes of ImageNet-1K to accommodate the 21,841 classes. Naturally, as the class classification is now 21,841, the output size of the last fully connected layer increases to 21,841. Thus, the last fully connected layer has more gradients to calculate for the increased number of weights and biases. Note that as the use of CNN models becomes more prevalent, classification into more classes should be expected in the future.

Figure 9 shows the results in a setup similar to Figure 7. Overall, the trend in performance is similar to ImageNet-1K with the following two key differences. First, the improvements brought about by RALP

Table 3: Speedup of RALP in workload consolidation

| | **Throughput** | | **Speedup** |
|---|---|---|---|
| | TensorFlow | RALP | |
| VGG11 × 8 | 91.3 | 651.8 | 7.14× |
| VGG11 × 4 | 120.6 | 592.2 | 4.91× |
| ResNet-50 × 4 | 351.9 | 386.9 | 1.10× |
| ResNet-50 × 8 | 400.5 | 409.8 | 1.02× |

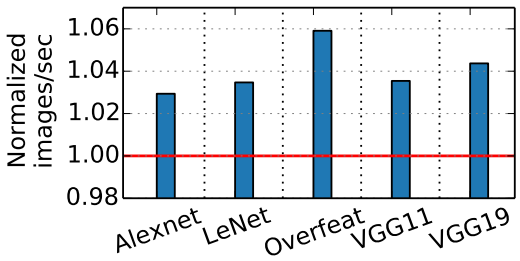



Figure 10: Performance using two PS workers normalized to using one PS worker

is substantially higher compared to both the baseline and Horovod (Note the *y*-axis is higher.). Second, even RALP-H beats Horovod in all cases. This means that even with slightly over half the resources, RALP is outperforming the Horovod.

### 3.4.4 Workload Consolidation

In this section, we consider the effect of RALP under workload consolidation, where we run multiple training jobs simultaneously in our shared cluster. For this study, we revert back to the ImageNet-1K dataset. We carry out evaluations under different sets of consolidated workloads as follows: (i) when the workload is homogeneous and consists of jobs that all benefit from RALP the most (i.e., RALP friendly jobs); (ii) when the workload is homogeneous and consists of jobs that do not benefit much from RALP (i.e., RALP unfriendly jobs); and (iii) when the workload is mixed with RALP friendly and unfriendly jobs. VGG11 and ResNet-50 are selected as the RALP friendly and unfriendly job, respectively. Each job runs on four GPUs, where workers are assigned three GPUs and the PS runs on a single GPU. One PS is deployed on one node to see the network's impact in workload consolidation. Thus, in our testbed, which has a total of 32 GPUs, we can consolidate up to 8 training jobs. Note that the TensorFlow setting is different from those of Sections 3.4.2 and 3.4.3 where the workers and PSes are set to equal numbers. This is to compare TensorFlow and RALP under the same configuration such that the difference between the two can be clearly identified.

Table 3 summaries the results showing that RALP achieves overall higher performance when the workload includes RALP friendly jobs. The most benefit comes from when the workload consists of RALP friendly jobs only. In particular, when running eight VGG11 jobs in the cluster, the average speedup with RALP is as much as 7.14 times, where speedup is calculated as the throughput with RALP

Table 4: Speedup of RALP in mixed workload

| | **Throughput** | | **Speedup** |
|---|---|---|---|
| | TensorFlow | RALP | |
| VGG11 × 4 | - | 605.2 | 5.02× |
| ResNet-50 × 4 | 384.3 | - | 1.09× |
| VGG11 × 4 | 122.9 | - | 1.02× |
| ResNet-50 × 4 | - | 352.1 | 1.00× |

divided by the throughput with TensorFlow. Moreover, we still observe the benefit when the workload is only partially RALP friendly, as can be seen in the second row in Table 3 where VGG11 exhibits 4.91× speedup. Note that ResNet-50, the unfriendly job, performance improves by 10% as well in the second row of the table. This indicates that under workload consolidation where network is actively shared, reduction in network communication results in aggregated benefits across all training jobs, both friendly and unfriendly, which is exactly what RALP brings about. Lastly, running only ResNet-50 jobs results in very small improvements, which is expected since RALP has limited benefit on ResNet-50 training itself.

Now we consider consolidating models with an intermix of approaches, that is, jobs with TensorFlow and RALP running simultaneously. Again, we make use of RALP friendly and unfriendly jobs, VGG11 and ResNet-50. Table 4 shows how the jobs are mixed as well as the results. In the first 2 rows, VGG11 and ResNet-50 are executed with RALP and TensorFlow, respectively, each of 4 jobs using 4 GPUs per job, while in the bottom 2 rows, the jobs are flipped to use the other schemes. The results reiterate the fact that the benefits from friendly jobs spill over even to unfriendly jobs resulting in aggregated benefits. Specifically, in the first two rows, VGG11 throughput improves by 5.02×, where this speedup is calculated by dividing the RALP result in Table 4 (605.2) by the VGG11×4 TensorFlow result in Table 3 (120.6). Interestingly, similar to the mixed workload results in Table 3, performance of the consolidated ResNet-50 improves by around 10% even when executed with TensorFlow as shown in the second row. In contrast, when VGG11 is run with TensorFlow, we see that performance of neither the friendly nor the unfriendly job improves since there is no network communication reduction.

### 3.4.5 Multiple PS Workers

In this section, we present results when using multiple PS workers implemented with pipeline parallelism [6]. In our cluster scale, we observe that using more than two PS workers is not beneficial, so in this experiment we limit RALP training to two PS workers and 30 workers (denoted by 2PS-30W). We compare 2PS-30W to using a single PS worker that exercises the same number of workers (denoted by 1PS-30W). For comparison we select five benchmarks (LeNet, Alexnet, Overfeat, VGG11, and VGG19), which consist of no more than three fully connected layers. In splitting the layers across two PS workers run in a pipeline, we take a simple heuristic approach that places the first layer, which is the largest, on the first PS worker and the remaining layers in the second PS worker.

Table 5: Speedup of RALP-N compared to BytePS

| | ImageNet-1K | | | ImageNet-22K | | |
|---|---|---|---|---|---|---|
| GPUs | 4 | 8 | 16 | 4 | 8 | 16 |
| Inception-v3 | 0.41 | 0.40 | 0.39 | 0.66 | 0.64 | 0.54 |
| ResNet-50 | 0.91 | 0.91 | 0.95 | 0.88 | 0.79 | 0.89 |
| VGG19 | 0.96 | 0.99 | 1.08 | 1.06 | 1.13 | 1.30 |

Figures 10 shows training performance of 2PS-30W normalized to that of 1PS-30W. We see that there is marginal performance improvement in 2PS-30W ranging from 2.9% to 5.9% (4% on average). Since 1PS-30W requires one GPU fewer than 2PS-30W, we also measure performance when exploiting the same total number of GPUs in the single PS worker configuration, i.e., there are one PS worker and 31 workers. In such case, the observed performance becomes very close to 2PS-30W.

#### 3.4.6 Comparison to BytePS

We evaluated with BytePS [66, 67], which is the state-of-the-art system for distributed DNN training. BytePS can take advantage of the cluster's spare CPU and network bandwidth resources to accelerate distributed DNN training jobs running on GPUs. Table 5 shows the speedup of RALP compared with BytePS according to the number of GPUs. Among the DNN models that can be executed in BytePS, three CNN models that can be compared with RALP were selected and evaluated for ImageNet-1K and 22K. RALP unfriendly jobs, Inception-v3 and ResNet-50, performed worse in RALP. However, VGG19, a RALP friendly job, had similar or slight performance improvements. In particular, RALP showed performance improvement in ImageNet-22K, which has a large parameter size. This is because RALP does not communicate large parameters.

### 3.5 Discussion and Summary

#### 3.5.1 Discussion

**Comparison to Stanza.** Our work shares some similarities with Stanza [68] in that both exploit the fact that data transfer is mostly attributed to memory-demand layers which incur insignificant computation (e.g., the fully connected layers in CNN models). Similar to RALP, Stanza proposes to colocate fully-connected layers with PS server to reduce network traffic. The key distinction is that RALP uses a model profiler so that we could enable layer splitting only when it is truly beneficial. (i) RALP can decide layer partitioning "dynamically" using a well-defined cost model, and (ii) RALP conducts layer partitioning transparently while Stanza has to change the program to use their technique. We showed in this paper that having such mechanism is critical since blindly applying layer splitting is not always beneficial.

**Model Partitioning.** RALP is partly inspired by OWT, which proposes to jointly use multiple parallelism schemes, especially for CNN using data parallelism on the convolutional layers and model parallelism on the fully-connected layers [24]. Similar to RALP, OWT exploits the disproportionate computation-communication characteristics that appear in the CNN model layers. Jia et al. propose layer-wise parallelism, enabling each individual layer to use an independent parallelism policy [56]. The scope of these studies is primarily on the use of intra-machine resources. Instead, our focus is on training on physically distributed machines, where such cost models proposed in OWT [24] and by Jia et al. [56] are not applicable. We thus proposed a cost model facilitated for model partitioning in a distributed setup and presented extensive experimental results in this study.

### 3.5.2 Summary

In Section III, we proposed a resource-aware layer placement (RALP) scheme to alleviate the network bottleneck and balance computation to improve performance for CNN distributed training. To achieve our goal, we first went through a thorough analysis of CNN characteristics considering the computational and communication needs of the layers that comprise CNN. We found that due to network communication which is on the critical path in distributed training, memory-demand layers of CNN should be placed in the parameter server machine to maximize performance. We provided a method to easily incorporate such observations in the TensorFlow framework. Experimental evaluations showed that many CNN models are able to reap significant performance improvements compared to the baseline TensorFlow framework as well as Horovod, a recently proposed state-of-the-art method.

# IV HetPipe: Pipelined Model Parallelism and Data Parallelism

For training DNN models, the use of GPU clusters is now commonplace. In such an environment, the use of heterogeneous GPUs is inevitable due to the short release cycle of new GPU architectures [34]. Moreover, several types of GPUs targeted for high-end servers, workstations, and desktops are being released for purchase [69–72]. Due to their cost-effectiveness, less expensive GPUs targeted for desktops and workstations, rather than high-end servers are also commonly used for machine learning training, especially for small and medium size clusters [73–78]. Due to the same reason, spot instances with different types of GPUs that are offered by cloud service providers are being used [34, 79, 80]. Table 6 shows the hardware specifications for four different types of GPUs, along with their market release years, that we have purchased in our institution in the short span of the last three years. Each, at the time of purchase, was (close to) state-of-the-art affordable with what budget we could muster. With technology advancing in such rapid pace, these systems have become outdated. Some of the systems have become old technologies that, individually, are unable to run large DNN models that are common today. Such situations with clusters of heterogeneous GPUs should now be commonplace.

There are benefits to enabling DNN training with heterogeneous resources. First, it allows for large model training with lower-class GPUs. While unable to train individually due to their limited resources, aggregated together, they may be used for training. For example, while one GeForce RTX 2060 GPU cannot be used to train ResNet-152 (with batch size = 32) [1] due to the memory constraint, training may be possible using multiple of these GPUs or perhaps in combination with other types of GPUs. These GPUs, which likely would have been retired, become usable, possibly used to create (virtual) workers that show similar performance as high-class GPUs. Second, low-class GPUs can be used to improve the performance of even high-class GPUs by incrementally adding on the resources of the (old) lower class systems to the (new) high-class systems. We call a group of aggregated GPUs that could satisfy the resource constraint and be used for training a *virtual worker*. Internally, such a virtual worker could leverage pipelined model parallelism (PMP) to process a minibatch, while externally, a number of virtual workers could leverage data parallelism (DP) for higher performance. Note that even in a homogeneous cluster where a single GPU can train a large model, exploiting PMP with DP can be beneficial, especially for a case that the amount of model parameters is huge and the cluster is connected via slow network. Also note that in a heterogeneous cluster, the performance of training a DNN model may improve with PMP and DP, compared to DP based on AllReduce communication, as the integration of PMP and DP may mitigate the problem of stragglers.

In Section IV, we explore the integration of PMP and DP to maximize the parallelism of DNN model training. In particular, we investigate a DNN model training system, which employs both PMP and DP, for a heterogeneous GPU cluster that possibly includes whimpy GPUs that, as a standalone, could not be used for training large models. Integrating DP to PMP may sound trivial, but in fact, it is quite challenging. In this setting, each virtual worker is continuously processing multiple minibatches in a pipelined manner and thus, all the virtual workers can be in different states. Thus, the key question here is, what weight version should be used by each virtual worker to synchronize with other virtual workers?

Table 6: Heterogeneous GPUs

| GPU | Year | Architecture | CUDA Core | Boost Clock (MHz) | Memory Size (GB) | Memory BW (GB/sec) |
|---|---|---|---|---|---|---|
| TITAN V | 2017 | Volta | 5120 | 1455 | 12 | 653 |
| TITAN RTX | 2018 | Turing | 4608 | 1770 | 24 | 672 |
| GeForce RTX 2060 | 2019 | Turing | 1920 | 1680 | 6 | 336 |
| Quadro P4000 | 2017 | Pascal | 1792 | 1480 | 8 | 243 |

Numerous questions need to be answered to answer this question: 1) How many new minibatches can start being processed while waiting for global updates from the parameter server? 2) Can synchronization occur at any point of processing the minibatches? 3) How can convergence be guaranteed when such synchronization occurs? 4) What version of parameters is used for the next minibatch while previous minibatches are still executing within each virtual worker? (This question is also considered to some extent in a prior work [26].) Furthermore, there are also many challenges that need to be overcome to ideally leverage a heterogeneous GPU cluster for DNN training: How are the heterogeneous GPUs to be divided and allocated into virtual workers? How do we reduce virtual worker stragglers when we consider DP? How do we partition the model to maximize the performance of PMP using heterogeneous GPUs?

While DP [13, 21–23], PMP [6, 26], and heterogeneity [31, 34, 81] for training have been considered separately, to the best of our knowledge, this is the first paper that tackles these issues together in attempting to answer *some* of the aforementioned questions. In this work, we design a DNN training system, *HetPipe* (*Het*erogeneous *Pipe*line), that integrates PMP of a virtual worker, which is composed of multiple (possibly whimpy) heterogeneous GPUs, with DP of virtual workers using parameter servers to enable and also speed up training of large models. HetPipe can aggregate heterogeneous resources from multiple GPUs to form a virtual worker such that the performance of each virtual worker is similar to each other, reducing the straggler problem. For HetPipe, we propose a novel parameter synchronization model, which we refer to as Wave Synchronous Parallel (WSP). WSP is adapted from the Stale Synchronous Parallel (SSP) model [33] to accommodate both PMP and DP for multiple virtual workers unlike existing synchronization models. We also prove the convergence of WSP. Note that while HetPipe would work in a homogeneous GPU cluster in training a large model that cannot be loaded into the memory of a single GPU, with the rapid turnaround of newer GPU architectures, it is more likely that one will end up with a cluster of heterogeneous GPUs. This is the environment that we target.

We implement HetPipe by modifying TensorFlow, a commonly used machine learning training system. We evaluate the performance of HetPipe for two DNN models using a heterogeneous GPU cluster composed of four different types of GPUs. Our experimental results demonstrate that the performance of HetPipe is better than that of the state-of-the-art DP via Horovod [22] that uses AllReduce communication [82]. This is because HetPipe mitigates the straggler problem, and also because it enables each

Table 7: Comparison of HetPipe with GPipe and PipeDream

| | GPipe | PipeDream | HetPipe |
|---|---|---|---|
| Heterogeneous Cluster Support | No | No | Yes |
| Target Large Model Training | Yes | No | Yes |
| Number of (Virtual) Workers | 1 | 1 | N |
| Data Parallelism | Extensible | Partition | Virtual Workers |
| Proof of Convergence | Analytical | Empirical | Analytical |

virtual worker and the parameter server to intra-communicate for all parameter updates, significantly reducing communication overhead. Compared to Horovod, the convergence of VGG-19 with a large parameter set to a desired accuracy becomes 49% faster, and that of ResNet-152 which is too big to be loaded in four whimpy GPUs in our cluster becomes 39% faster by using all the GPUs (including whimpy ones).

Strategies to leverage PMP have been explored in previous studies [6, 26, 83, 84]. Compared to these, our study makes forward strides in three aspects. First, we generalize PMP of a virtual worker to be used together with DP of virtual workers, increasing the parallelism of DNN model training. Consequently, this results in speeding up training. Second, we consider a heterogeneous GPU cluster, which allows the use of GPUs, which otherwise, could not be used for training. Finally, we present a parameter synchronization model that guarantees convergence, for training models using PMP with DP. We provide a more in-depth comparative discussion on these studies in Section 4.1.

### 4.1 Model Parallelism and Pipeline Execution

This PMP strategy has been investigated in previous studies [6,26]. PipeDream exploits PMP of a single virtual worker to avoid the parameter communication overhead of DP [26]. Considering only homogeneous GPUs, when PipeDream partitions a model into stages to maximize pipeline performance, it does not take into account the memory requirement of a GPU that depends on the stage of a pipeline. Thus, PipeDream processes a limited number of minibatches, which is large enough to saturate the pipeline, to reduce memory overhead. PipeDream also provides a form of DP, but it considers DP within a virtual worker to speed up the execution of lagging layers. No proof of single pipeline convergence is provided in PipeDream. Note that without a parameter synchronization model such as WSP, it is not possible to properly run DP over multiple PipeDream virtual workers via parameter servers or AllReduce communication.

GPipe is a scheme that leverages PMP of a single virtual worker to support large DNN models, also in a homogeneous GPU cluster [6]. In GPipe, a minibatch is divided into multiple *microbatches* that are injected into the pipeline. Using the same weights, GPipe executes the forward passes for all the microbatches, and then executes the backward passes for them. When the backward pass of the last microbatch is done, it updates the weights all together for the minibatch. GPipe incurs frequent pipeline

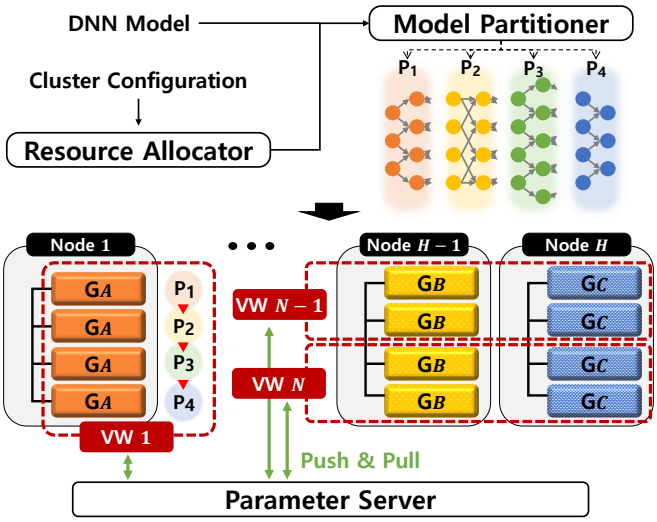



Figure 11: System architecture (VW: Virtual Worker)

flushes, possibly resulting in low GPU utilization [26]. In GPipe, DP of multiple virtual workers can be done using existing synchronization schemes like BSP as a virtual worker processes one minibatch at a time. GPipe saves on GPU memory by recomputing the activations again in the backward pass instead of keeping the activations computed in the forward pass in memory. We do not use this optimization though there are no fundamental reasons forbidding it. A comparison of HetPipe with previous studies is given in Table 7.

## 4.2 System Overview

The system that we propose focuses on training a large DNN model in a *heterogeneous GPU cluster* composed of various types of GPUs that have different computation capability and memory capacity. In such settings, for some types of GPUs in the cluster, the DNN model of interest may be too large to be loaded into the memory of a single GPU. The system that we propose leverages both pipelined model parallelism (PMP) and data parallelism (DP) to enable training of such large DNN models and, in the process, enhance performance as well as the utilization of the heterogeneous GPU resources of the cluster.

Figure 11 shows the architecture of the proposed cluster system composed of $H$ nodes. Each node comprises a homogeneous set of GPUs, but the GPUs (and memory capacity) of the nodes themselves can be heterogeneous. Two key novelties exist in this architecture. First, DP is supported through a notion of a *virtual worker* (VW), which consists of $k$, possibly heterogeneous, GPUs, and encapsulates the notion of a worker in typical DNN systems. That is, a virtual worker is used to train the DNN model. In Figure 11, note that there are $N$ virtual workers with 4 GPUs each, that is, $k = 4$, and that the GPUs comprising the virtual worker may be different for each virtual worker. While in this paper we consider $k$ to be constant for each virtual worker, our design does not restrain it to be so; this is simply a choice we make for simplicity. The key aspect here is that a virtual worker allows DP by aggregating GPUs possibly even when individual GPUs may be resource limited.

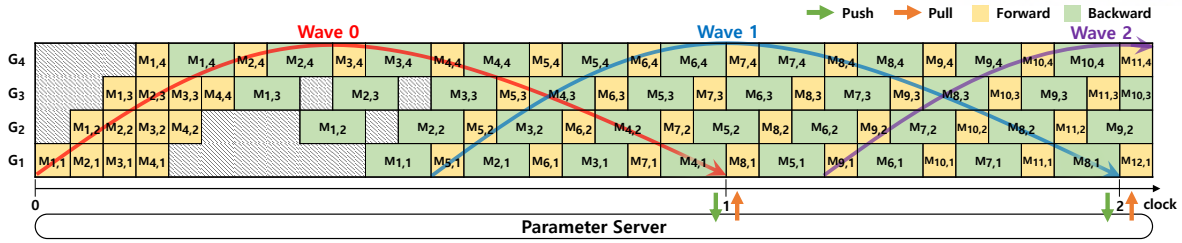



Figure 12: Pipeline execution of minibatches where $M_{p,k}$ indicates the execution of a minibatch $p$ in partition $k$, which is executed in GPU$_k$ and the yellow and green colors indicate the forward and backward passes, respectively.

The second novelty is that each virtual worker processes each minibatch based on model parallelism, in a pipelined manner, to fully utilize the GPU resources, as shown in Figure 12, to accommodate large DNN models. While PMP has been proposed before (which we compare in Section 4.1), to the best of our knowledge, we are the first to present PMP in a heterogeneous setting. We refer to our system as *HetPipe* as it is *het*erogeneous, in GPUs, across and, possibly, within virtual workers and makes use of *pipe*lining in virtual workers for resource efficiency.

To train DNN models based on pipelined model parallelism in virtual workers, the *resource allocator* first assigns $k$ GPUs to each virtual worker based on a resource allocation policy (which will be discussed in Section 4.6.1). Note that for allocating the heterogeneous GPUs to the virtual workers, the resource allocation policy must consider several factors such as the performance of individual GPUs as well as the communication overhead caused by sending activations and gradients within a virtual worker, and synchronizing the weights among the virtual workers and the parameter server. Then, for the given DNN model and allocated $k$ GPUs, the *model partitioner* divides the model into $k$ partitions for the virtual worker such that the performance of the pipeline executed in the virtual worker can be maximized.

As any typical DP, multiple virtual workers must periodically synchronize the global parameters via parameter servers or AllReduce communication; in HetPipe, parameter servers are used to maintain the global weights. Each virtual worker has a local copy of the global weights and periodically synchronizes the weights with the parameter server. Evidently, when managing the weights within a virtual worker and across virtual workers, two types of staleness, *local staleness* and *global staleness*, need to be permitted to improve the performance of DNN training. Local staleness refers to staleness within a virtual worker. As each virtual worker processes minibatches in a pipelined manner, there are multiple minibatches that are being processed in parallel. Thus, staleness is inevitable as weights seen by a minibatch may not reflect the updates of all of its previous minibatches.

Global staleness, on the other hand, is similar to the staleness notion introduced by Ho et al. [33]. That is, the system needs to reduce communication overhead between the parameter server and (virtual) workers, and, in our case, also mitigate the synchronization overhead caused by possibly heterogeneous virtual workers. Therefore, similarly to SSP [33], each virtual worker should be allowed to proceed training without querying the global weights for every minibatch, unless its local copy is so old such that there are too many missing recent updates made by other virtual workers. Note that such staleness condition is set by the user [33].

For our system, we propose the *Wave Synchronous Parallel* (WSP) model to synchronize the weights. A *wave* is a sequence of minibatches that are processed concurrently in a virtual worker. Let the number of minibatches in a wave be $N_m$. Within a wave, processing of the $i$-th minibatch is allowed to proceed without waiting for the preceding minibatchs $i'$ to be completed, where $1 < i \leq N_m$ and $1 \leq i' < i$. That is, there is no dependency among the weights used by minibatches in the same wave. As the virtual worker does not enforce the updates even from the first minibatch in a wave to be reflected in the weights used by the last minibatch, the local staleness threshold in WSP is $N_m - 1$. Moreover, *each virtual worker only pushes the aggregated updates from all the minibatches in a wave, instead of for every minibatch, to the parameter server.* This results in considerable reduction in communication overhead.

Note that HetPipe uses parameter servers, which may incur synchronization and communication overhead. However, HetPipe mitigates such overhead by permitting global staleness among virtual workers and executing the pipeline in each virtual worker such that it continues to process minibatches that have already been injected while waiting for the parameter update. We believe HetPipe can be further optimized by taking decentralized approaches, but leave this for future work.

### 4.3 Pipelined Model Parallelism Within a VW

**Number of Minibatches in the Pipeline.** In our system, each virtual worker processes up to $N_m$ minibatches concurrently in a pipeline manner so that the executions of the minibatches can overlap. Given a DNN model and $k$ GPUs, the maximum number of minibatches executed concurrently in the virtual worker, $Max_m$, is basically determined by the memory requirement for training the model. For a model that requires a huge amount of memory for output activations and weights, $Max_m$ may be less than $k$. Note that in such cases, the utilization of each GPU is unlikely to be high.

$N_m$, the actual number of minibatches in the pipeline will be $N_m \leq Max_m$ and basically determined by considering the throughput of the pipeline. Note that $N_m$ must be the same in every virtual worker, and thus, $N_m$ is set to the minimum $Max_m$ among all the virtual workers. $N_m$ will affect the local staleness that we discuss later in this section.

**Model Partitioning.** To train a DNN model, a set of $k$ GPUs is allocated to a virtual worker by a resource allocation policy, which we discuss in Section 4.6.1. For now, let us assume that $k$, the number of possibly heterogeneous GPUs, and $N_m$ are given. Then, a partitioning algorithm is employed to divide multiple layers of the model into $k$ partitions, assigning them to the $k$ different GPUs. The goal of the partitioning algorithm is to maximize the performance of the pipeline, while satisfying the memory requirement of each partition to process $N_m$ minibatches.

In particular, in this study, for memory, we consider the fact that the actual memory requirement will vary depending on the stage of the pipeline that the GPU is used for. For example, contrast $\text{GPU}_4$ and $\text{GPU}_1$ in Figure 12. $\text{GPU}_4$, the GPU that handles the last stage of the pipeline, handles only one minibatch at a time and is immediately done with the minibatch as exemplified by the yellow (forward pass) and green (backward pass) $\text{M}_{i,4}$ pairs for $i = 1, 2, ...$, that are side-by-side. In contrast, for $\text{GPU}_1$, the yellow and green $\text{M}_{i,1}$ pairs are far apart, meaning that the forward pass $\text{M}_{i,1}$ needs to hold up memory

until the backward pass $M_{i,1}$ is finished with its execution. Thus, with $GPU_1$, the memory requirement is high as it needs to hold on to the results of the forward pass for all stages of the pipeline. This variance in memory requirement is considered in partitioning the layers.

Execution time must also be considered when partitioning the layers. To do so, we calculate the execution time of a partition to be the sum of the computation time of all the layers in the partition and the communication time needed for receiving the activations (in the forward pass) and local gradients (in the backward pass). Our partitioning algorithm attempts to minimize the maximum execution time of the partitions within the bounds of satisfying the memory requirement.

**Partition Scheduling.** Once the partition is set, the partitions need to be scheduled for each of the GPUs. Each $GPU_q$ responsible for partition $q$ may have multiple forward pass and backward pass tasks to schedule at a time. Each GPU schedules a task by enforcing the following conditions:

1. A forward pass task for a minibatch $p$ will be executed only after a forward pass task for every minibatch $p'$ is done where $1 \leq p' < p$.
2. Similarly, a backward pass task for a minibatch $p$ will be executed only after a backward pass task for every minibatch $p'$ is done where $1 \leq p' < p$.
3. Among multiple forward and backward pass tasks, a FIFO scheduling policy is used.

Note that in the last partition, for a minibatch, processing a forward pass immediately followed by a backward pass is executed as a single task.

**Considering Staleness.** Given the description of pipelining, the question of staleness of weights used needs to be considered. That is, as a minibatch is scheduled, it may be that the layers are not using the most up-to-date weights. For example, in Figure 12, when the forward pass $M_{2,1}$, the second minibatch, begins to be processed, it must use stale weights as the first minibatch has not completed and hence, the changes in the weights due to the first minibatch have not yet been appropriately reflected, which is in contrast with typical processing where minibatches are processed one at a time. We now discuss how this staleness issue is considered.

Let *local staleness* be the maximum number of missing updates from the most recent minibatches that is allowed for a minibatch to proceed in a virtual worker. As training with $N_m$ minibatches can proceed in parallel in a virtual worker, the local staleness threshold, $s_{local}$, is determined as $N_m - 1$, where $1 \leq N_m \leq Max_m$. If $N_m = 1$, the behavior is exactly the same as naive model parallelism. Larger $N_m$ may improve the performance (i.e., throughput) of the pipeline as a larger number of concurrent minibatches are executed, but local staleness increases, possibly affecting the convergence of training. In a real setting, typically, $N_m$ will not be large enough to affect convergence as it will be bounded by the total amount of GPU memory of a virtual worker.

Such local staleness also exists in PipeDream [26]. As PipeDream basically employs weight stashing that uses the latest version of weights available on each partition to execute the forward pass of a minibatch, a different version of weights is used across partitions for the same minibatch. Unfortunately, PipeDream only shows empirical evidence of convergence when weight stashing is used. Note

that PipeDream also discusses vertical sync, which is similar to HetPipe, but it excludes vertical sync in its evaluations [26].

Now let $w_p$ be the weights used by minibatch $p$. Then, initially, we can assume that $w_0$, the initial version of weights, is given to the virtual worker. Then, the first $(s_{local}+1)$ minibatches are processed in a pipelined manner with $w_0 = w_1 = \cdots = w_{s_{local}} = w_{s_{local}+1}$.

To accommodate staleness in our system, when processing of minibatch $p$ completes, the virtual worker updates the local version of the weights, $w_{local}$ as $w_{local} = w_{local} + u_p$, where $u_p$ is the updates computed by processing minibatch $p$. Therefore, in HetPipe, weights are not updated layer by layer and $w_{local}$ is a consistent version of weights across partitions. When the virtual worker starts to process a new minibatch, it makes use of the latest value of $w_{local}$ without waiting for the other minibatches to update their weights. For example, once the virtual worker is done for minibatch 1 and updates $w_{local}$ with $u_1$, it will start to process minibatch $s_{local}+2$ by using the updated weights without waiting for minibatches 2 up to $s_{local}+1$ to be completed. Similarly, when the virtual worker is done with minibatch $s_{local}+1$ and updates $w_{local}$ with $u_{s_{local}+1}$, it will start to process minibatch $2\times(s_{local}+1)$ without waiting for the previous most recent $s_{local}$ minibatches to be completed. Therefore, except for the initial minibatches 1 to $s_{local}+1$, for minibatch $p$ the virtual worker will use the version of the weights that reflects (at least) all the local updates from minibatches 1 to $p-(s_{local}+1)$. Note that for every minibatch $p$, $w_p$ must be kept in GPU memory until the backward pass for $p$ is executed.

Note that staleness in SSP is caused by the different processing speed of minibatches among multiple workers. Thus, in SSP, staleness is used as a means to reduce the synchronization and communication overhead. However, local staleness in HetPipe is caused inherently as minibatches are processed in a pipelined manner within a virtual worker.

### 4.4 Data Parallelism with Multiple VWs

In this section, we discuss data parallelism (DP) with virtual workers. The first and foremost observation of DP being supported with virtual workers is that the virtual workers may be composed of (whimpy) heterogeneous GPUs. While it is well known that DP helps expedite DNN execution, DP, in typical systems, is not possible if individual GPUs, that is, workers, do not have sufficient resources to handle the DNN model, in particular, large DNNs. By allowing a virtual worker to be composed of multiple GPUs that are lacking in resources, our system allows DP even with whimpy GPUs. The other key observation in properly supporting DP with virtual workers is that each virtual worker now retains local staleness as discussed in Section 4.3. Making sure that, despite such individual staleness, we understand and show that the results obtained from DP among virtual workers (globally) converge is an important issue that must be addressed. The rest of the section elaborates on this matter.

**Workings of WSP.** As stated in the system overview, HetPipe uses parameter servers. We assume that such synchronization occurs in *clock* units, a notion taken from SSP [33]. Precisely, a clock unit is defined as the progress of completing one wave. Recall from Section 4.2 (and Figure 12) that a wave is a

sequence of $s_{local}+1$ minibatches concurrently executed such that a virtual worker is allowed to process a later minibatch in a wave without updates from an earlier minibatch in the same wave.

Similarly to SSP (which, however, considers the staleness of weights only in DP), each virtual worker maintains a local clock $c_{local}$, while the parameter server maintains a global clock $c_{global}$, which holds the minimum $c_{local}$ value of all the virtual workers. Initially, the local clocks and the global clock are 0. At the end of every clock $c$, each virtual worker completes the execution of all the minibatches in wave $c$. At this point, the virtual worker computes the aggregated updates from minibatch $c \times (s_{local}+1)+1$ to minibatch $(c+1) \times (s_{local}+1)$ and pushes the updates ũ to the parameter server. We see that, similar to in SSP [33], ũ is synchronized with a clock value $c$. For example, as shown in Figure 12 where $s_{local}=3$, at the end of clock 0, the virtual worker pushes the aggregated updates of wave 0, which is composed of minibatches from 1 to 4, and at the end of clock 1, the aggregated updates of wave 1, which is composed of minibatches from 5 to 8, and so on. It is important to note that in WSP, the virtual worker pushes ũ to the parameter server for every wave, instead of pushing ũ for every minibatch, which will significantly reduce the communication overhead.

When the parameter server receives the updates ũ from the virtual worker, the parameter server updates the global version of the weights as $w_{global} = w_{global} +$ ũ. Note that the parameter server updates its $c_{global}$ to $c+1$ only after every virtual worker has pushed the aggregated updates of wave $c$.

In WSP, each virtual worker is allowed to proceed training without retrieving the global weights for every wave. Thus, the virtual worker may use a weight version that, from a global standpoint, may be stale, as the most recent updates received by the parameter servers may not be reflected in its local version of the weights. We discuss how global staleness among the virtual workers is bounded.

**Global Staleness Bound.** Let *clock distance* be the difference in $c_{local}$ between the fastest and slowest virtual workers in the system. Therefore, a virtual worker with local clock $c$, where $c \geq D+1$, must use a version of the weights that includes all the (aggregated) updates from wave 0 up to $c-D-1$. Also, the weight version may include some recent global updates from other virtual workers and some recent local updates within the virtual worker beyond wave $c-D-1$.

When a virtual worker pulls the global weights at the end of clock $c$ to maintain this distance, it may need to wait for other virtual workers to push their updates upon completion of wave $c-D$. However, while a virtual worker waits for other virtual workers to possibly catch up at the end of clock $c$, local processing is allowed to proceed with $s_{local}$ minibatches of wave $c+1$ as the minibatches are executed in a pipelined manner. Take, for example, the case when $D=0$ and $s_{local}=3$ in Figure 13 (and Figure 12). As a virtual worker, VW1, completes minibatch 4, it computes the aggregated updates ũ for wave 0 (composed of minibatches 1 to 4) and pushes ũ to the parameter server. VW1 now waits for the other virtual workers to complete wave 0 before proceeding with minibatch 8. However, note that as shown in the figure, VW1 has already started to process minibatches 5, 6 and 7, which belong to wave 1, while its local clock is still 0. Similarly, once it completes minibatch 8, it pushes the aggregated updates ũ for wave 1 (composed of minibatches 5 to 8) to the parameter server; in the meantime, it has already started processing minibatches 9, 10, and 11, which belong to wave 2, while its clock is still 1.

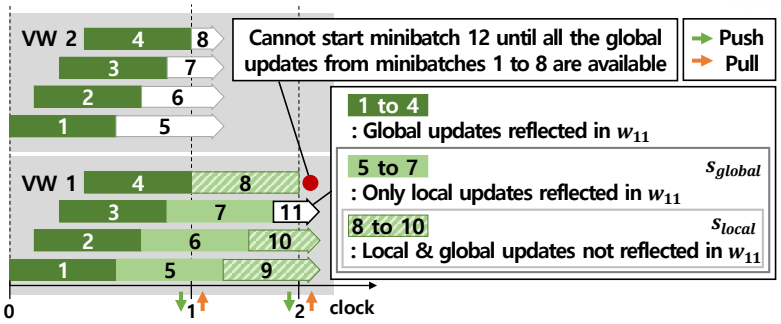



Figure 13: Local and global staleness with WSP

Note that this processing of local minibatches in the virtual worker does not violate the local staleness bound. Note also that when $D = 0$, each virtual worker must wait for each other at the end of every clock to synchronize the weights for every wave, which is BSP-like behavior with pipelined execution in each virtual worker.

Now let us define the *global staleness bound*, $s_{global}$, to be the maximum number of missing updates from the most recent minibatches, *globally computed by all the other virtual workers in the system*, that is allowed for a minibatch to proceed in a virtual worker. We want to identify $s_{global}$ based on our discussion so far. This will allow each virtual worker to determine whether it can proceed with its current minibatch.

Initially, all virtual workers start processing the first $(D+1)$ waves without querying the global weights from the parameter server. Furthermore, they can start to process up to $s_{local}$ minibatches of the next wave before receiving the global weights that include the recent updates as discussed above. Therefore, for those initial minibatches, the virtual worker uses $w_0$ or a weight version that may include some recent local updates.

For any minibatch $p$ thereafter, that is, where $p > (D+1) \times (s_{local}+1) + s_{local}$, $p$ must use a weight version that reflects, at the very least, all the global updates from all the other virtual workers from minibatch 1 to minibatch $p-(s_{global}+1)$, where $s_{global} = (D+1) \times (s_{local}+1) + s_{local} - 1$. The first term of this equation is due to the fact that a virtual worker is allowed to proceed with the next $(D+1)$ waves (i.e., $(D+1) \times (s_{local}+1)$ minibatches), and the second term is due to the additional $s_{local}$ minibatches that can be started because of pipelined execution. Continuing with the example in Figure 13, where $D = 0$ and $s_{local} = 3$, VW1 proceeds the training of minibatch 11 without the global and/or local updates from wave 1 (minibatches 5 to 8) or the two local updates from minibatches 9 and 10 (i.e., having $s_{global} = 6$). Thus, it must have a version of the weights that includes all the global updates from minibatches 1 to 4. Actually, the weight version used for minibatch 11 includes three local updates from minibatches 5, 6, and 7, along with all the global updates from wave 0. In case of minibatch 12, it cannot start the training until global updates up to minibatch 8 are received.

**Convergence Analysis.** We provide theoretical proof of the WSP model. The full proof is omitted due to outside the scope of this dissertation but can be found in [85]. Our theoretical results are similar

with existing work on non pipelined version of staleness update [33, 34]. However, we reflect the new characteristics of distributed pipeline staleness update.

## 4.5 Partitioning Algorithm

Recall that the goal of our partitioning algorithm is to minimize the maximum execution time of the partitions within the bounds of satisfying the memory requirement. To obtain a performance model to predict the execution time of each layer of a model in a heterogeneous GPU, we first profile the DNN model on each of the different types of GPUs in a cluster, where we measure the computation time of each layer of the model. For GPU memory usage, we measure the usage of each layer (by using the logging feature of TensorFlow) on only one GPU type (as it is roughly the same for all GPU types). For profiling the memory usage on a whimpy node, we measure the memory usage of each layer using a small batch size and then multiply it for the target batch size. To compute the memory requirement for a given partition, we take into account the total memory usage to store the data to process the layers as well as the maximum number of minibatches concurrently assigned to the partition.

For communication time between layers in the model, we first derive the amount of input data for each layer in the forward and backward pass from the model graph. For the given data size, we predict intra-node communication based on the PCI-e bandwidth, then multiply it by a scaling-down constant (which is similarly done in Paleo [86]), since in practice, it is not possible to utilize the peak bandwidth. The scaling-down constant is derived by running a synthetic model that sends various sizes of data from one GPU to another GPU in the same node. For inter-node communication (via InfiniBand), we use linear regression to estimate the communication time for the given data size. To build a prediction model, we collect 27 samples by training two DNN models, used in our experiments, with arbitrary partitions. Note that in this work, the heterogeneity of network performance such as slow network links is not considered (as in [31]). However, for such cases, we can extend our partitioning algorithm to consider different network performance between two nodes when estimating the communication time. Also, a model that estimates the memory requirement for each stage more accurately will be helpful in partitioning a DNN model in a more balanced manner.

To find the best partitions of a DNN model, we make use of CPLEX, which is an optimizer for solving linear programming problems [87]. The memory requirement for each partition on the pipeline to support $N_m$ concurrent minibatches is provided as a constraint to the optimizer. The algorithm will return partitions for a model with a certain batch size only if it finds partitions that meet the memory requirement for the given GPUs. Also, the optimizer checks all the different orders of the given heterogeneous GPUs for a single virtual worker to partition and place layers of the DNN model on them.

## 4.6 Experimental Results

### 4.6.1 Methodology

**Heterogeneous GPU Cluster.** In our experiments, we use four nodes with two Intel Xeon Octa-core E5-2620 v4 processors (2.10 GHz) connected via InfiniBand (56 Gbps). Each node has 64 GB memory

and 4 homogeneous GPUs. Each node is configured with a different type of GPU as shown in Table 6. Thus, the total number of GPUs in our cluster is 16. Each GPU is equipped with PCIe-3×16 (15.75 GB/s). Ubuntu 16.04 LTS with Linux kernel version 4.4 is used. We implement HetPipe based on the WSP model by modifying TensorFlow 1.12 version[1] with CUDA 10.0 and cuDNN 7.4.

**DNN Models and Datasets.** Our main performance metric is throughput (images/second) of training a DNN model. We use ResNet-152 [1], and VGG-19 [40] with ImageNet [88]. For each DNN model, batch size of 32 is used. For all other hyperparameters, we use the default settings as specified in the benchmark [89] of ResNet-152 and VGG-19.

**Resource Allocation for Virtual Workers.** Given any heterogeneous GPU cluster, there can be many ways of allocating the resources to the multiple virtual workers. For our experiments, we consider allocation policies within the bounds of our platform. Thus, given the 16 GPUs, HetPipe employs four virtual workers, each of which is configured with four GPUs, along the following three allocation policies.

- **Node Partition (NP):** This policy assigns a node per virtual worker. Thus, each virtual worker is composed of homogeneous GPUs. Consequently, as the nodes are heterogeneous, partitioning of layers for a DNN model is different for each virtual worker. NP results in minimum communication overhead within each virtual worker as communication between GPUs occurs within the same node via PCI-e, rather than across multiple nodes where communication is via InfiniBand. On the other hand, as the performance of each virtual worker varies, a straggler may degrade performance with DP.

- **Equal Distribution (ED):** This policy evenly distributes GPUs from each node to every virtual worker. Thus, every virtual worker is assigned four different GPUs, but every virtual worker has the exact same resources. Thus, model partitioning is the same, and thus, performance will be the same across the virtual workers, which mitigates the straggler problem. However, ED results in high communication overhead within each virtual worker.

- **Hybrid Distribution (HD):** This policy is a hybrid of NP and ED. For our cluster, a combination of two GPU types are allocated to each virtual worker such that their performances in terms of aggregated computation capability and amount of GPU memory are similar to each other. This choice is made to mitigate the straggler problem while reducing the communication overhead within each virtual worker. As, in terms of computation power, `V` > `R` > `G` > `Q` and, in terms of the amount of the GPU memory, `R` > `V` > `Q` > `G`, two virtual workers are allocated `VVQQ`, while the other two are allocated `RRGG`, where `V`, `R`, `G` and `Q` refers to TITAN V, TITAN RTX, GeForce RTX 2060, and Quadro P4000, respectively.

Table 8 shows the resource allocation of each virtual worker for the three resource allocation policies.

[1]Modified LOC is ∼1.5K in the TensorFlow framework and TensorFlow benchmark codes, where most features are added as independent functions.

Table 8: Resource allocation for the three policies considered

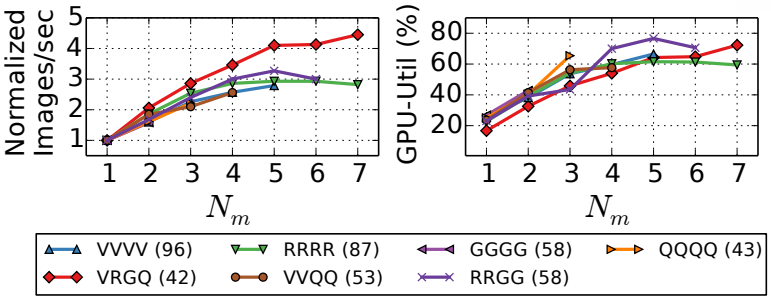
| | Node Partition | Equal Distribution | Hybrid Distribution |
|---|---|---|---|
| VW1 | VVVV | VRGQ | VVQQ |
| VW2 | RRRR | VRGQ | VVQQ |
| VW3 | GGGG | VRGQ | RRGG |
| VW4 | QQQQ | VRGQ | RRGG |

**Parameter Placement.** In our experiments, for DP, we locate the parameter servers, each of which only handles a portion of the model parameters, over all the nodes. For the *default* placement policy, which can be used with all three of our resource allocation policies, we place layers of the model in round-robin fashion over all the parameter servers as in TensorFlow [90]. For ED, however, another policy is possible, which we refer to as 'ED-local'. With 'ED-local', we place the layers of a partition on the parameter server running on the same node, incurring no actual network traffic across the nodes for parameter synchronization. This is possible as the same partition of the model can be assigned *locally* to the GPU on the same node for every virtual worker. For all results reported hereafter, the 'default' policy is used, except for 'ED-local'.

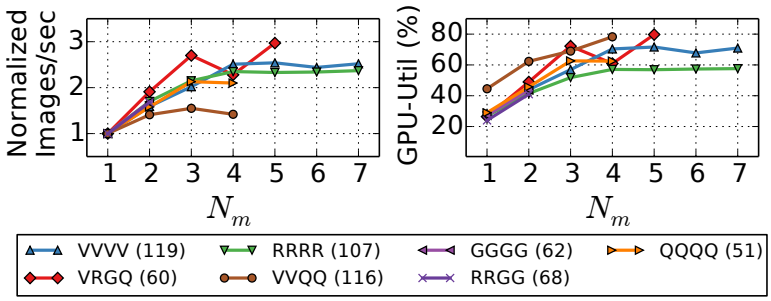
### 4.6.2 Performance of a Single Virtual Worker

We first investigate the performance of the 7 different individual virtual workers that are possible according to the allocation schemes in Table 8. Figure 14 shows the throughput over various values of $N_m$, which is the number of minibatches executed concurrently, in the virtual worker normalized to that of when $N_m = 1$ and the maximum average GPU utilization among the four partitions for ResNet-152 and VGG-19. The numbers shown (in the box) along with the allocation policy are the absolute throughput (images/sec) when $N_m = 1$. Note that some results for larger $N_m$ are not shown. This is because the GPU memory cannot accommodate such situations and hence, cannot be run.

From the results, we can see that as $N_m$ increases, normalized throughput of a virtual worker as well as the maximum GPU utilization generally increases. Note that, though not shown, the total GPU memory utilization tends to increase as $N_m$ increases. However, depending on the resource allocation scheme (which results in different partitions of a model) as well as the DNN model, the effect of having larger $N_m$ varies. When a virtual worker is configured with homogeneous GPUs, the average GPU utilization of each partition is similar to each other. However, when it is configured with heterogeneous GPUs, there is a tendency that the GPU utilization of the first or last partition is higher than those of the other partitions. For this configuration, different computation capabilities and memory capacity of the GPUs are considered when partitioning a model. As it is possible that only a small number of layers are assigned to some GPUs, the overall GPU utilization may turn out to be low.




(a) ResNet-152




(b) VGG-19

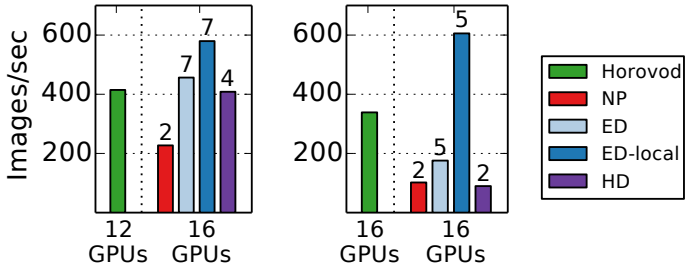
Figure 14: Normalized throughput and the maximum average GPU utilization among partitions in a single virtual worker for various resource allocation policies as $N_m$ is varied. The number in parenthesis is absolute throughput (images/sec) when $N_m = 1$ (which is equivalent to the naive MP) for each policy.

### 4.6.3 Performance of Multiple Virtual Workers

Figure 15 shows the throughput of training each model with the three resource allocation policies, where "Horovod" indicates the state-of-the-art DP via Horovod that uses AllReduce communication[2]. In these experiments, for each resource allocation policy, $N_m$ is set such that performance is maximized while every virtual worker uses the same value of $N_m$ as this is the assumption behind HetPipe. For ResNet-152, the whole model is too large to be loaded into a single GPU with `G` type, and thus, Horovod uses only 12 GPUs.

The results in Figure 15 show that the performance of DNN training is strongly affected by how heterogeneous GPUs are allocated to virtual workers. From the results, we can make the following observations: First, for VGG-19 whose parameter size is 548MB, the performance of Horovod, which reduces communication overhead for parameter synchronization, is better than those of NP, ED, and HD. However, for ResNet-152 whose parameter size is 230MB, ED and HD, which utilize virtual workers with similar performance, show a bit better or similar performance to Horovod (with 12 GPUs). Second, with NP, training performance of ResNet-152 and VGG-19 is low as $N_m$ is bounded by the virtual worker with the smallest GPU memory. Third, with ED-local, intra-communication occurs between each GPU and the parameter server, significantly reducing communication overhead across the nodes, especially

[2]We use the same minibatch size for all workers of Horovod as the minibatch size is one of the critical factors to the final performance of a trained DNN and adaptive batch sizing will affect convergence [91].




(a) ResNet-152 (b) VGG-19

Figure 15: Performance with the three allocation policies when D=0 (The number on bar represents $N_m$)

Table 9: Performance improvement of adding whimpy GPUs (The number in parenthesis presents the total number of concurrent minibatches in HetPipe)

| Model | Single GPU [V] | Method | 4 GPUs 4[V] | 8 GPUs 4[VR] | 12 GPUs 4[VRQ] | 16 GPUs 4[VRQG] |
|---|---|---|---|---|---|---|
| VGG-19 | 159 | Horovod | 164 | 205 | 265 | 339 |
| | | PipeDream | ERROR[3] | ERROR[3] | ERROR[3] | 286 |
| | | HetPipe | 300 (5) | 530 (16) | 572 (20) | 606 (20) |
| ResNet-152 | 112 | Horovod | 233 | 353 | 415 | X |
| | | PipeDream | 238 | 382 | 224 | 225 |
| | | HetPipe | 256 (5) | 516 (20) | 538 (24) | 580 (28) |

for VGG-19, the model with a large parameter set. For VGG-19, the amount of data transferred across the nodes per minibatch with ED-local (i.e., 103MB) is much smaller than that with Horovod (i.e., 515MB). Thus, the performance of ED-local (which also mitigates the straggler problem) is 1.8× higher than Horovod. For ResNet-152, the amount of data transferred with ED-local (i.e., 298MB) is larger than that with Horovod (i.e., 211MB) because the sizes of output activations to be sent between partitions are large, even though the parameter size is relatively small. However, the throughput of ED-local is still 40% higher than Horovod. This is because Hetpipe allows each virtual worker to process a large number of minibatches concurrently. Compared to NP and HD, ED-local (or ED) usually has larger $N_m$ in each virtual worker, improving throughput.

Next, we investigate how the throughput is improved when whimpy GPUs are additionally used for training. Table 9 shows the throughput of VGG-19 and ResNet-152 when DP via Horovod, PMP via PipeDream and HetPipe with ED-local are used over different sets of heterogeneous GPUs, and also when a single `V` GPU is used. For these experiments, HetPipe is configured to use four virtual workers, except for '4 GPUs' where a single virtual worker is used. In the table, the number and type of GPUs used for each experiment are also given. From the results, we can see that the performance of both

[3]In the VGG-19 model, some GPU options had a communication error. It was discussed in PipeDream's open-source issues, and so far the error has not been resolved [92].

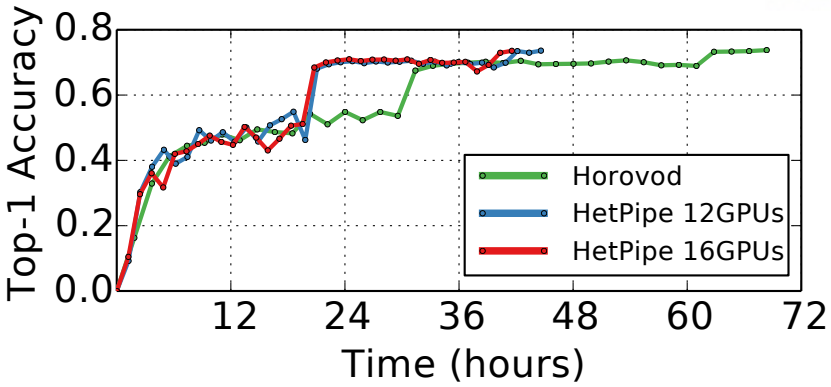



Figure 16: ResNet-152 top-1 accuracy

Horovod and HetPipe increases when additional whimpy GPUs are used for training. With additional GPUs, HetPipe can increase the total number of concurrent minibatches processed, having up to 2.3 times speedup. The pipeDream [26] of ResNet-152 was faster than Horovod and slower than HetPipe when using 4 or 8 GPUs. Since the PipeDream's partitioning algorithm does not consider heterogeneous GPUs, so adding whimpy GPUs slows it down. A detailed comparison to PipeDream is described in Section 4.7.1 Discussion. This scenario can be thought of as an answer to when new, higher end nodes are purchased, but one does not know what to do with existing nodes. The results show that making use of the whimpy systems allows for faster training of larger models.

### 4.6.4 Convergence

Our HetPipe based on the WSP model is guaranteed to converge as proven inin [85]. In this section, we analyze the convergence performance of HetPipe with ED-local using ResNet-152 and VGG-19. For our experiments, the desired target accuracy of ResNet-152 and VGG-19 is 74% and 67%, respectively.

Figure 16 shows the top-1 accuracy of ResNet-152 with Horovod (12 GPUs), HetPipe (12 GPUs), and HetPipe (16 GPUs), where $D$ is set to 0 for HetPipe. For the experiments with 12 GPUs, the 4 `G` type GPUs are not used. When the same set of GPUs are used, convergence with HetPipe is 35% faster than that of Horovod by reducing the straggler problem in a heterogeneous environment and exploiting both PMP and DP. Furthermore, by adding four more whimpy `G` GPUs, HetPipe improves training performance even more, converging faster than Horovod by 39%.

Figure 17 shows the top-1 accuracy of VGG-19 with Horovod and HetPipe as we vary $D$ to 0, 4, and 32. For the experiments, all 16 GPUs are used. The figure shows that convergence with the BSP-like configuration (i.e., $D = 0$) of HetPipe is roughly 29% faster than that with Horovod. As we increase $D$ to 4, the straggler effect is mitigated and the communication overhead due to parameter synchronization is reduced. Thus, convergence is faster by 28% and 49% compared to $D = 0$ and Horovod, respectively. In this experiment with ED-local (where the training speed of each virtual worker is similar), when $D$ becomes very large (i.e., 32), the throughput remains similar but the convergence performance degrades by 4.7%, compared to $D = 4$. This is because it is unlikely that the clock distance between the fastest and slowest virtual workers becomes as large as 32, but higher global staleness can degrade the convergence

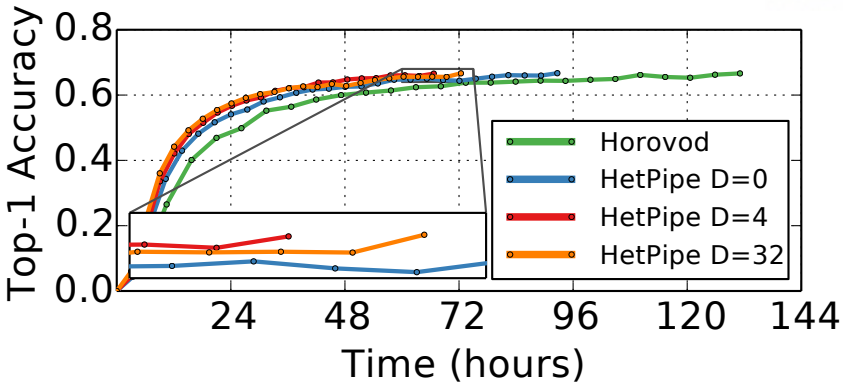



Figure 17: VGG-19 top-1 accuracy

performance (similarly discussed in [33]). Note that though not shown, using larger $D$ has a greater effect for HetPipe with NP, ED and HD resource allocation, and the different resource allocations only affect the set of heterogeneous GPUs used for each virtual worker and do not affect the convergence behavior.

We also analyze the synchronization overhead as $D$ is varied. We find that as $D$ increases, the waiting time of a virtual worker to receive the updated global weights decreases. In our experiments, the average waiting time with $D = 4$ is found to be 62% of that with $D = 0$. Furthermore, the actual idle time is only 18% of the waiting time as the virtual worker can continue to proceed in the pipeline while waiting.

## 4.7 Discussion and Summary

### 4.7.1 Discussion

**Comparison to PipeDream.** PipeDream [26], which is the closest related study, optimizes PMP of a single virtual worker, only employing DP for lagging layers within a virtual worker in homogeneous environments. To be adapted to heterogeneous environments, its partitioning algorithm must be extended to consider the different performance and memory sizes of heterogeneous GPUs, various orders of heterogeneous nodes used for a pipeline, and the memory requirement of the GPUs for partitions.

We run the training of ResNet-152 using PipeDream, which is implemented on PyTorch [93], in our heterogeneous GPU cluster described in Section 4.6.1. Since the partitioning algorithm does not consider heterogeneous GPUs, for each GPU type, we profile ResNet-152, then generate partitions of the model assuming that our cluster is configured with homogeneous GPUs with that type, and finally, measure the throughput of PipeDream with the partitions. All the computed configurations of the pipeline result in a large number of (i.e., 12 or 14) partitions. For example, with `Q`, the configuration is 4-2-1-1-1-1-1-1-1-1-1-1 indicating that the model is divided into 12 partitions where the first partition is executed by four GPUs with DP, the second one is executed by two GPUs with DP, and so on. For these configurations, we run experiments with various orders of the four different nodes and test using several batch sizes. (Note that we could not run training for some configurations due to out of memory errors.) The best throughput measured using PipeDream is 158. Recall that the throughputs of Horovod (with 12 GPUs) and HetPipe

are 415 and 580, respectively. In this case, the performance of PipeDream for ResNet-152 is found to be low as a large number of partitions cause high network overhead, in addition to the sub-optimal partitions. Therefore, with PMP alone (i.e., single virtual worker), the performance benefit may become limited when a model is divided into numerous partitions. Instead of increasing partitions, running DP with multiple virtual workers like HetPipe can improve the parallelism of training and further improve performance in such cases.

**Effect of Imbalanced Partitions.** Our partitioning algorithm attempts to balance partitions while satisfying the memory requirements. However, depending on the DNN model, computed partitions may be imbalanced. For example, for a model composed of a small number of layers, if one layer takes much longer to execute compared to other layers, the partitions may end up having different execution times. In this case, the performance of the pipeline will be degraded as in any other pipeline-based systems. Note that running DP for the slow partition to have a similar processing rate across all the partitions like PipeDream [26] will be a possible extension of HetPipe.

### 4.7.2 Summary

In Section IV, we presented a DNN training system, HetPipe, that integrates pipelined model parallelism with data parallelism. Leveraging multiple virtual workers, each of which consists of multiple, possibly whimpy, heterogeneous GPUs, HetPipe makes it possible to efficiently train large DNN models. We proved that HetPipe converges and presented results showing the fast convergence of DNN models with HetPipe.

# V RAP: Parallelizing Dataset Pre-Processing

For higher accuracy, DNN training requires a large amount of data. In the DNN model, a pre-processing process is required to train a dataset. Pre-processing is used when converting datasets into binary forms to improve training speed or converting raw datasets into input forms necessary for training specific DNN models. For example, in TensorFlow, the dataset can be pre-processed as a sequence of binary records type TFRecord to load data efficiently [94]. On the other hand, in the DLRM model, such as the DNN-based recommendation, pre-processing is needed to train a raw dataset and convert it into an appropriate input type [95]. Most of the studies have focused on DNN training efficiency, but there have not been many studies on the pre-processing of the dataset.

We focus on parallelization of pre-processing for DNN-based recommendations to increase the efficiency of dataset pre-processing. The reasons why dataset pre-processing for the DNN-based recommendation system is essential are as follows. (1) It is a fundamental transformation for training: Pre-processing for efficient load is optional, but dataset pre-processing for DNN-based recommendation is essential for training. (2) Pre-processing procedure is not simple: In order to make the load efficient, processing different from pre-processing, which converts to binary format or bundles data, is required. In the dataset of the DNN-based recommendation model, the chronological order of feature indexing is essential. This makes it challenging to apply parallelism. This is covered in detail in Section 5.1. (3) It takes a considerable time: For reasons (2), parallelization has not been applied so far, and it takes longer than processing other datasets. For example, it takes 112 minutes to pre-process the 138 GB ImageNet-1K [52] dataset, which is widely used for image classification, into TFRecord. However, it takes 139 hours to pre-process the 1 TB Criteo Terabyte [96] dataset for DNN-based recommendation. (Both experiments used Machine-H of Table 11 of Section 5.4) It takes longer, even considering the size of the dataset.

We propose a Resource-Adaptive Pre-processing (RAP) scheme that improving performance while considering the computing resources. To develop RAP, we go through an analysis of dataset pre-processing steps, where we discuss three insights that we enlighten for parallelizing dataset pre-processing. We found a parallelizable way to solve the existing problem. In particular, we develop resource-adaptive parallelism that can monitor resources and adjust the degree of parallelism according to the condition to maximize resource use. Furthermore, RAP supports pre-processing on distributed nodes. Our experimental results show that, by applying resource-adaptive parallelism, the pre-processing time is reduced by up to 93% compared to the traditional serial execution. Utilizing four distributed nodes can reduce time by up to 98%.

The rest of Section V is organized as follows. In the next section, we discuss dataset pre-processing for DNN-based recommendation system as background. In Section 5.2, we discuss three insights that we enlighten for parallelizing dataset pre-processing. Based on these insights, we present the RAP in Section 5.3. In Section 5.4, we present experimental results, and finally, in Section 5.5, we conclude with a summary.

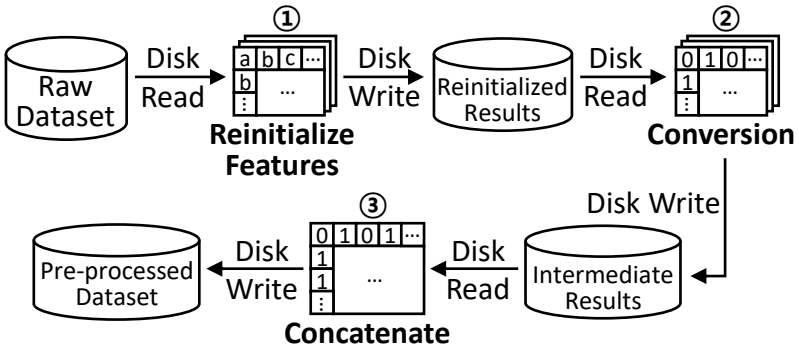



Figure 18: Pre-processing procedure

## 5.1 Dataset Pre-Processing for DNN-based Recommendation System

Recommendation systems are widely used in click-through rate (CTR) prediction models that are based on user's preferences. In particular, the Deep Neural Network (DNN)-based recommendation systems have attracted much attention from social media, advertisements, e-commerce and various services. Facebook's DLRM [97] model is used in social media and Google's Wide&Deep [98] model is used in Google Play Store. Alibaba's DIN [99] and DIEN [100] models are used in e-commerce. In addition, there are many DNN-based recommendation systems, including NCF [101] and MT-Wide&Deep [102]. These models take the history behavior sequence of users and items' categorical features as inputs to train and predict CTR.

The DNN-based recommendation system has a cyclical life cycle of dataset pre-processing, model training and inference services. The recommendation system model should be retrained daily or weekly with new interactive data for timely personalization inference [103]. Until now, most studies have concentrated on development of models for training and inference performance [104–108]. However, dataset pre-processing is required before the training (including retraining) process, and this pre-processing time is not one to be neglected. For example, when training the DLRM model with the Criteo Terabyte dataset [96], pre-processing took 107 hours, while training time took 3.5 hours to converge to 96.74% accuracy on our high-end system (Machine-H of Table 11 of Section 5.4 with DLRM executed with the default settings [109]). Pre-processing takes a significant amount of time.

To help develop new models various datasets are available for recommendation system training and to use these datasets, a pre-processing procedure is required to filter the data and transform it into the form required as input. The dataset consists of logs about categorical features. For example, the MovieLens [110] dataset contains categorical features for movies such the genre, rating, user age, etc. The Criteo Terabyte [96], which consists of 4.3 billion training data log samples for 24 days, and Alibaba Dataset [100], which consists of two billion samples for two weeks, have click feedback information and categorical features such as the gender of the clicker and the type of merchandise that could be used for advertisements. Criteo Terabyte is public, while Alibaba Dataset is a private industrial dataset. YouTube creates hundreds of billions of user logs every day [102]. While for one dataset, pre-processing only needs to be done once, periodic data pre-processing is required for new interactive data retraining.

Pre-processing for DNN-based Recommendation system consists of three time-consuming steps. Figure 18 shows the pre-processing procedure of DNN-based recommendation system. The first step is to reinitialize the features, which basically takes the categorical features and re-inserts them appropriately to the data structures for pre-processing. This requires reading the entire raw dataset, from typically a set of files, in chronological order [① in Figure 18] as the pre-processed dataset must retain the order of the raw dataset and as recommendation systems require training according to the chronological order of the dataset. Before the files with the reinitialized results are stored in storage, a feature indexing procedure is executed creating a file that retains information on the existence of features in the dataset that are derived from the reinitialized data. Traditionally, this is done as the data items are processed in chronological order. Since the datasets' chronological order is essential, it was challenging to parallelize this traditional pre-processing step. For this reason, even when there were many CPU cores, serial execution was performed making use of only a single core.

In the second step, which is the conversion step, the files that contain reinitialized categorical features that are in unicode string format are converted into files that contain purely integer format [②]. These converted files are then compressed and stored back to storage. This step is memory and I/O intensive as format conversion and data compression requires considerable memory and saving the intermediate results causes writes to storage.

In the final step, all the files created in Step 2 are concatenated into a single file, and then compressed. Note that the concatenation must be done in the chronological order of the files [③]. The resulting file is used for training the recommendation system.

Note that the three steps may all be combined to be done in a single step and need not be separated out. However, in practice, it is done in this manner for "checkpointing" purposes just in case a failure occurs during the pre-processing phase. This is because, as mentioned previously, the pre-processing phase can take tens to hundreds of hours depending on the dataset size. The last step of the three steps is not mandatory. Therefore, in this study, we will deal with Steps 1 and 2.

## 5.2 Three Insights to Parallelizing Dataset Pre-Processing

Traditionally, the dataset pre-processing stage was conducted in a sequential manner. For large datasets such as Terabyte, this resulted in over a hundred hours of pre-processing time even on state-of-the-art manycore systems. In this section, we discuss three insights that we enlighten for parallelizing dataset pre-processing to expedite the process. We discuss them one by one.

**Insight 1:** The first insight in parallelizing dataset pre-processing is based on the observation that, while it is important that the data used for training be in chronological order, the data themselves are not chronological dependent on each other. That is, data obtained in, say day 10, has no bearing on the data of day 11 or any thereafter, allowing the pre-processing of each data item to be done in parallel. For example, the Criteo Terabyte [96] dataset is organized in 24 file units where each file represents a day's worth of data. Hereafter, we will refer to each of the units in which data is read and written to storage as the I/O unit, or simply, the *unit*. Note that in this case, a unit are not necessarily of the same size. If

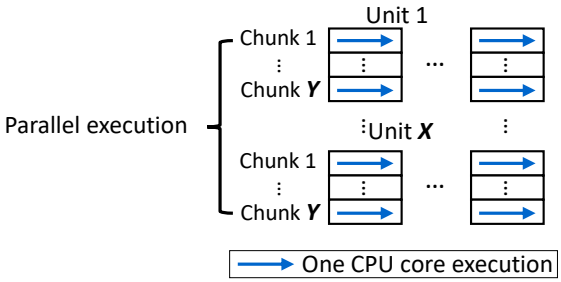



Figure 19: The number of units $X$, and the number of chunks $Y$ of parallel execution

we set the parallel degree with a naive heuristic, it would be this unit. We can parallelize with this unit if we apply parallelization with a naive heuristic. We refer to this as *unit parallelism*. Unit parallelism can be employed at both Steps 1 and 2. However, in Step 1, there is the feature indexing procedure that requires a chronologically ordered, that is, sequential processing of the entire dataset to generate a feature indexing file.

**Insight 2:** The second insight for parallelism is that as each unit may be processed in parallel, recursively, the data within the unit may also be processed in parallel. That is, once the unit is brought into memory, they can be partitioned into multiple chunks of data, and the chunks can be processed in parallel. Note that the feature indexing procedure in Step 1 can also be improved by first performing the procedure on each unit in parallel and then merging them together into a single file. Here we can see that the number of units and the number of chunks to be processed at one time are different depending on the resource of the computing node. When a compute node has more CPU cores than the total number of units, it is practical to process it by partitioning it into chunks. Also, considering the memory size of the computing node, it may not be possible to process all units at once. As shown in Figure 19, the number of units executed in parallel at one time can be expressed as $X$, and the number of chunks in each unit can be expressed as $Y$. It is crucial to determine $X$ and $Y$ to be processed at once, depending on the situation of the computing resource. Section 5.3 describes the Resource-Adaptive Pre-processing (RAP) system that determining $X$ and $Y$ to suit the computing resource situation.

**Insight 3:** Another insight for possible parallelism is to make use of pipelined parallelism. Pipelined parallelism overlaps execution among the pre-processing components. For example, recall that in Step 1, the feature indexing procedure was performed in sequential manner. We can pipeline parallelize this procedure by overlapping the reinitializing procedure with the feature indexing procedure. Even for Step 2, where feature indexing is nonexistent, reinitialization and writing to storage can be pipelined. The size of the chunk affects the efficiency of the pipeline. If the chunk size is large, the processing time is long, and the write is short, so the pipeline overlap will not work well. On the other hand, if the chunk size is too tiny, writes will occur more frequently, causing bottlenecks in I/O. It needs an appropriate chunk size Y considering I/O bandwidth. We will cover this in the next section.

### 5.3 Resource-Adaptive Parallelizing Dataset Pre-Processing

In this section, we discuss Resource-Adaptive Pre-processing (RAP). RAP determines the degree of parallelism according to the situation of computing resources. We describe the *resource manager*, which manages and monitors computing resources and determines the degree of parallelism. We also describe distributed pre-processing using multiple nodes.

We need to find the number of units and the number of chunks to make the most of the resource. Corresponds to the value of $(X,Y)$ in Figure 19. We consider the number of CPU cores, the size of the memory, and the I/O bandwidth of the computing resource. The value of $(X,Y)$ affects the utilization of each resource. (1) When the number of CPU cores is $C$, $C = X \times Y$ must be satisfied to utilize the entire cores. (2) When the size of memory used to process one unit is $M_{usage}$, to process $X$ units, the total memory size of $M$ must satisfy the following. $M \geq X \times M_{usage}$. (3) System I/O bandwidth ($I/O_{bw}$) must satisfy the following in order to maximize disk utilization. $I/O_{bw} \geq X \times Y \times I/O_{usage}$. RAP has a resource manager to monitor resources and determine the values of $(X,Y)$.

**Resource Manager.** $M_{usage}$ and $I/O_{usage}$ can be measured as pre-processing execute. RAP initially sets $(X,Y)$ to $(1,C)$. This is executed by dividing the chunk by the number of CPU cores for one unit. When pre-processing the first unit, the resource manager can check the resource utilization for one unit and utilize the entire core. During the first $(1,C)$ run, RAP's resource manager measures $M_{usage}$ and $I/O_{usage}$ when pre-processing one unit. The resource manager monitors the resource utilization every second and has a max value for $M_{usage}$ and a max value and mean value for $I/O_{usage}$. After the first unit execution of $(1,C)$, each time the unit ends, adjust the $(X,Y)$ value to suit the resource's state.

In order to better utilize the Insight 3 pipeline parallel, the value of $I/O_{bw}$ should be maximized. If $I/O_{usage}$ is low, the chunk size should be reduced by increasing the Y value. This is because the smaller the chunk size, the more frequent I/O. Conversely, if the bottleneck due to high I/O usage, reducing the Y value will alleviate the I/O bottleneck and enable effective pipeline execution. The resource manager doubles Y to increase the utilization of $I/O_{usage}$ when the mean value of $I/O_{usage}$ when running the previous unit is less than 90% of the max value. On the other hand, if it exceeds 90%, it is found that the use of I/O is a bottleneck, and Y is reduced in half. Naturally, the value of X is also determined by the number of cores constraint $(C = X \times Y)$. This is to utilize the entire core while controlling the running chunk. $M_{usage}$ checks if $M \geq X \times M_{usage}$ is satisfied when the value of $X$ increases. If there is insufficient system memory, the resource manager will not be able to increase the value of $X$. If $X$ does not increase, the Y value does not become smaller.

Algorithm 1 expresses the conditions for determining $X$ and $Y$ in the resource manager as an algorithm. For example, if the utility of $I/O_{usage}$ is low after running the first unit with $(1,C)$, then running the following unit will be $(1/2, 2 \times C)$. On the other hand, when $I/O_{usage}$ is a bottleneck, it will be $(2, C/2)$ if satisfied after checking $M \geq X \times M_{usage}$. If there is not enough space in memory, it is kept as $(1,C)$.

**Algorithm 1** Algorithm for determining the number of units and chunks in resource manager.

X: Number of units to be parallelized, Y: Number of chunks to be parallelized

```
X = 1, Y = C (Number of cores)
if I/O_usage−max * 0.9 > I/O_usage−mean:
    Y = Y * 2,   X = X / 2
else:
    if M ≥ M_max * 2:
        Y = Y / 2,   X = X * 2
```

Table 10: Write size at each step

| Dataset | Raw Dataset Size | Reinitialize Step | Conversion Step |
|---|---|---|---|
| Terabyte | 1.0 TB | 247.0 GB | 218.7 GB |
| Kaggle | 10.4 GB | 3.1 GB | 2.2 GB |

**Parallelizing Distributed Pre-Processing.** RAP is also capable of distributed processing. Each computing node pre-processes the raw dataset on different units. Each node has its resource manager, and pre-processing is performed in the same manner as when pre-processing in a single node. However, it should be noted that, as explained in Insight 1, feature indexing in Step 1 must retain the chronological order. After Step 1 is finished, each node transmits the feature indexing result to the master node (node 0) and waits. When the master node receives files from all nodes, it sends the entire feature indexing file in chronological order to all nodes. Each node proceeds to Step 2 after receiving feature indexing from the master node.

## 5.4 Performance Evaluation

**Testbed.** Table 11 presents the specifications of the two machines used in the evaluation. Machine-H is a modern **H**igh-end server while Machine-L is a relatively old and **L**ow-end server. For the experiments, hyper-threading was enabled (maximum of 80 and 32 cores for Machine-H and Machine-L, respectively) and turbo boost (the numbers in parentheses in the CPU specification of the table indicating the maximum turbo frequency). We use Python 3.7.7 for pre-processing on both machines.

**Dataset.** For evaluation, we select the Criteo Terabyte [96] and Kaggle [111] datasets of different sizes. The raw datasets of Terabyte and Kaggle are provided in 24 and 7 file units, respectively. This dataset was selected because it is available for various recommendation system models (e.g., DLRM, Wide&Deep, DIN, DIEN, NCF and MT-Wide&Deep). Table 10 shows the size of the two datasets' initial raw dataset and write size at each step, while the size in the previous column is the amount read in that particular step. For example, in Step 1 of the Terabyte dataset, 1TB of data is read and 247GB of data are written out.

Table 11: Machine specifications

| | Machine-**H** | Machine-**L** |
|---|---|---|
| CPU | 2 × Intel Xeon 20-core Gold 6230 (3.90 GHz) | 2 × Intel Xeon Octa-core E5-2620 v4 (3.00 GHz) |
| Memory | 1 TB | 256 GB |
| Storage | 4 TB SSD | 4 TB SSD |
| Network | 100 Gbps InfiniBand | 56 Gbps InfiniBand |
| Software | Ubuntu 20.04 Linux kernel 5.4 | Ubuntu 16.04 Linux kernel 4.15 |

**Implementation.** We implement the pre-processor parallelization insights in Facebook's DLRM data generation and loading code [109]. Each of the three parallelization insights are implemented at each of the reinitialize and conversion steps using Python's multiprocessing library. For Insight 3, overlapped execution was instigated using the Event API function provided in the multiprocessing library. In Step 1, the reinitialize step of the next unit and the feature indexing procedure of the current unit are overlapped, while for Step 2, the conversion step, the execution of the next unit and the writing of the intermediate results of the current unit are overlapped. Finally, parallel pre-processing can be invoked simply by adding an execution option with the parallelization mode name to be used in each step. Distributed pre-processing was implemented through python's shell command embedding. The feature indexing procedure is saved as a file and transmitted to each node for communication.

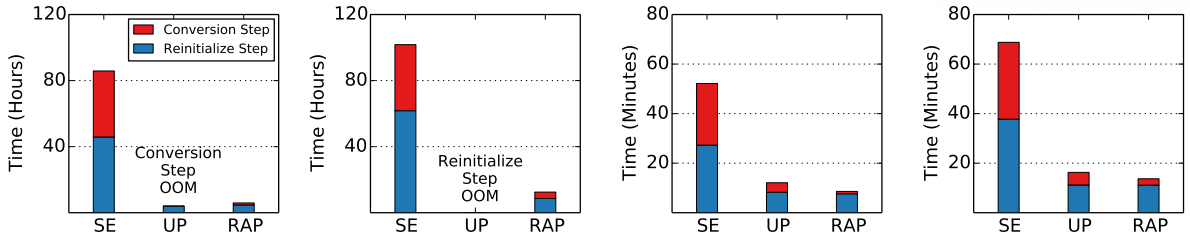
Insight 1, unit parallelism, has already been open-sourced and integrated within the DLRM data generation and loading code [112].

### 5.4.1 Overall performance

**Performance metric.** We measure the execution times of Steps 1 and 2 that represent the reinitialize and conversion step, respectively. There are other miscellaneous operations, such as counting the number of lines of data and dividing the dataset into training and test data, that comprise part of the entire pre-processing sequence, but we do not report them as they form a relatively small part of the operation. Compare with serial execution (SE), unit parallel (UP), and resource-adaptive pre-processing (RAP).

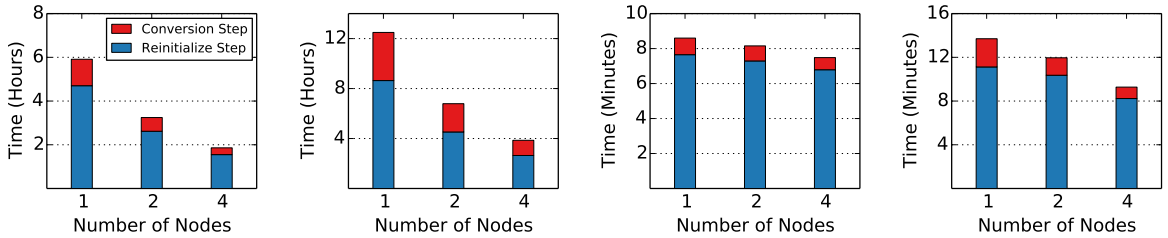
Figure 20 shows the pre-processing time of each dataset on two machines. In the unit parallel of the Terabyte dataset in Machine-H in Figure 20 (a), the reinitialize step finished the fastest, but the pre-processing could not be completed because OOM occurred in the conversion step. However, in the case of RAP, all pre-processing was completed by efficiently utilizing resources. Machine-L in Figure 20 (b) has a smaller memory than Machine-H, so in the unit parallel, OOM occurred in the reinitialize step. Pre-processing the Terabyte dataset, in Machine-H and Machine-L, RAP has 93% and 89% faster results than serial execution, respectively.

In smaller Kaggle datasets in Figure 20 (c) and (d), the unit parallel is faster than serial execution, and RAP is faster than unit parallel on both machines. There is a further speedup in the low-end server




(a) Terabyte (Machine-**H**) (b) Terabyte (Machine-**L**) (c) Kaggle (Machine-**H**) (d) Kaggle (Machine-**L**)

Figure 20: Pre-processing time of Terabyte and Kaggle dataset on two machines. SE, UP, and RAP represent serial execution, unit parallel, and resource-adaptive parallel, respectively.




(a) Terabyte (Machine-**H**) (b) Terabyte (Machine-**L**) (c) Kaggle (Machine-**H**) (d) Kaggle (Machine-**L**)

Figure 21: Distributed pre-processing time of Terabyte and Kaggle dataset on two machines using RAP.

Machine-L. Pre-processing the Kaggle dataset, in Machine-H and Machine-L, RAP has 84% and 80% faster results than serial execution, respectively.

### 5.4.2 Distributed Dataset Pre-processing Performance

Figure 21 shows the time of distributed pre-processing for each machine and dataset when using RAP. The *x*-axis represents the number of machines with the same specification. When the number of nodes is 1, it corresponds to RAP in Figure 20. In (a) and (b) of Figure 21, in the Terabyte dataset, the pre-processing time decreases almost linearly in both machines. On the other hand, in the case of the Kaggle dataset in Figure 21 (c) and (d), the decrease in time is not significant as more nodes are used. This is because it is a small dataset for distributed pre-processing.

## 5.5 Summary

In Section V, we analyzed the pre-processing of the DNN-based recommendation system with three insights. Based on the insights, the resource-adaptive parallelization of pre-processing was developed. Furthermore, it has extended to distributed pre-processing. By applying four distributed pre-processing, the pre-processing that took 112 hours for serial execution was reduced by up to 97% to 4 hours.

# VI Related Works

Numerous studies have been conducted to improve DNN performance. In this section, we discuss various methods that have been proposed to improving performance during model training.

**Efficient Distributed Training.** Horovod proposes an efficient communication method, namely, ring-reduce to improve network communication performance [22]. We compared RALP and HetPipe with Horovod extensively and found that our two systems scale better. Other work such as Project Adam [11] and Poseidon [17] decomposes gradients in the fully connected layer while reducing communication traffic. RALP and HetPipe could make use of these techniques to improve efficiency of parallel execution of the PS worker. Lastly, Das *et al.* propose an algorithm for hybrid data and model parallelism [113]. Some studies demonstrate the effectiveness of distributed training on other aspects. GeePS uses GPU memory as a cache for manipulating large scale training [16]. Parameter Hub proposes a software design that provides a streamlined gradient processing pipeline [114]. Awan *et al.* propose a pipelined chain design for the MPI_Bcast collective operation [115]. Xue *et al.* optimize RPC performance over RDMA network to accelerate distributed deep learning [116]. Hierarchical AllReduce performs the AllReduce operation in two levels [117]. This technique does not solve the straggler problem in a heterogeneous GPU cluster, as master GPUs in the second level will have different GPU types. BlueConnect is an efficient AllReduce communication library considering heterogeneous networks [118]; unfortunately, it also cannot handle stragglers caused by heterogeneous GPUs.

**Hybrid Parallelism.** There have been earlier efforts to employ DP and/or MP for model training. Project Adam uses both DP and MP to train machine learning models on CPUs [11]. Pal et al. combine DP and MP in a similar way as our system, but do not consider pipelining nor heterogeneous GPUs [119]. STRADS leverages MP to address the issues of uneven convergence of parameters and parameter dependencies [84]. FlexFlow considers utilizing parallelism in various dimensions such as sample, operator, attribute and parameters to maximize parallelization performance [42]. Bounded staleness has been explored where Jiang et al. present heterogeneity-aware parameter synchronization algorithms based on the SSP model [34], while Cui et al. analyze the effects of bounded staleness [120].

**Pipelining Training.** Pipelining has been leveraged to improve the performance of machine learning systems [6, 23, 26, 83, 121]. A pipelining scheme is employed to handle expensive backpropagation [83]. Pipe-SGD pipelines the processing of a minibatch to hide communication time in AllReduce based systems [23]. A weight prediction technique is proposed to address the staleness issue in pipelined model parallelism [121]. Detailed comparisons of HetPipe with PipeDream [26] and GPipe [6] are provided in Section 4.1. Note that the feature of overlapping computation and communication, presented in PipeDream [26], will also improve the performance of our system. PipeDream employs the one-forward-one-backward scheduling algorithm for pipeline execution. Sophisticated schedulers that consider various factors such as heterogeneous configurations, the number of partitions, and the number

of concurrent minibatches within a virtual worker, can potentially improve the performance of HetPipe. Techniques to optimize learning rates have been studied [122], which can also be applied to HetPipe to help converge faster. After HetPipe's work, various studies of pipelining parallelization for DNN training have been proposed. DAPPLE proposes a new parallelization strategy planner for solving partitioning and placement problems [123]. PipeMare enables fine-grained asynchronous updates during pipeline parallel execution without sacrificing memory [124].

**Model Partitioning.** As the model size becomes larger, model training needs to be efficiently done in resource-constrained environments. Strads shows that well-scheduled model parallelism using parameter convergence can perform better than normal training in some common machine learning applications [84]. Strads proposes a scheme called scheduled model parallelism that schedules parameter updates to speed up model convergence [84]. PipeDream partitions layers of model to place them on all available GPUs while achieving its dual goal of balancing computation and minimizing communication [26]. It reduces communication for large DNNs relative to data-parallel training. Gpipe partitions a model across different TPUs and automatically splits a mini-batch of training examples into smaller batch sizes called micro-batches [6]. The partitioning algorithm is heuristic-based minimizing the variance of computation in the pipeline stages.

**Decentralized Training.** Decentralized training systems that consider heterogeneous environments have also been studied [31, 125]. However, these techniques do not consider integration of DP with PMP, which allows support for large models that do not fit into single GPU memory. In AD-PSGD, once a mini-batch is processed, a worker updates the parameters by averaging them with only one neighbor which is randomly selected [31]. This is done asynchronously, allowing faster workers to continue. In theory, the convergence rate of AD-PSGD is the same as SGD. In principle, the contribution of AD-PSGD is orthogonal with the contributions of HetPipe in that we can extend our HetPipe further by adapting the idea of asynchronous decentralized update in AD-PSGD when there is a bottleneck in the parameter server. When it comes to the experimental evaluations, the performance of AD-PSGD is evaluated for DNN models whose sizes are 1MB, 60MB, and 100MB, which are smaller than the models we consider in HetPipe. For a decentralized training system, Hop [125] considers the bounded staleness and backup workers, and uses CIFAR-10 for performance evaluation on a CNN model.

**Gradient Optimization.** One of the reasons for poor scalability in distributed training is network communication bottleneck. In effect, a popular approach to mitigating network communication cost is reducing actual data to transfer without restructuring training workflow as done in RALP. There are two methods for such data-driven optimization, namely, gradient quantization and gradient sparsification. Gradient quantization focuses on manipulating gradients to reduce the amount of data transfers using ternary value [126], random discrete values [127], and low bitwidth [128]. Gradient sparsification focuses on dropping some gradients out from transfers through discarding gradients randomly [129] or the smallest gradients [130].

# VII Conclusion

In this dissertation, we investigated how to apply parallelism according to the DNN model and computing resources. Two DNN distributed training parallelization systems RAPL and HetPipe, and one dataset pre-processing parallelization system, RAP was covered. This is because the effective parallelization scheme is different depending on the characteristics of the model and the condition of the computing resources. Hybrid parallelization was constructed with partial data parallelism or integrated pipelined model parallelism and data parallelism. Also, pre-processing of the dataset applies resource-adaptive parallelization.

As the DNN models and datasets of DNNs grow, distributed parallelism becomes essential. This dissertation allows us understanding whether the parallelization approach affects the DNN training performance. The importance of the parallelization approach was analyzed according to the characteristics of the DNN models and the computing resource condition. Hybrid parallelism approaches have been shown to provide more suitable and effective parallelism.

# References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2012.

[3] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury, "Deep Neural Networks for Acoustic Modeling in Speech Recognition," *IEEE Signal Processing Magazine*, 2012.

[4] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research*, 2003.

[5] R. Collobert and J. Weston, "A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2008.

[6] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2019.

[7] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized Evolution for Image Classifier Architecture Search," in *Proceedings of the Conference on Association for the Advancement of Artificial Intelligence (AAAI)*, 2019.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2017.

[9] M. Wang, C.-c. Huang, and J. Li, "Supporting Very Large Models using Automatic Dataflow Graph Partitioning," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2019.

[10] T. Jin and S. Hong, "Split-CNN: Splitting Window-based Operations in Convolutional Neural Networks for Memory System Optimization," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[11] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an Efficient and Scalable Deep Learning Training System," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large Scale Distributed Deep Networks," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2012.

[13] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling Distributed Machine Learning with the Parameter Server," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[14] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A New Platform for Distributed Machine Learning on Big Data," in *Proceedings of the Conference on Knowledge Discovery and Data Mining (KDD)*, 2015.

[15] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-Caffe: Co-designing MPI Runtimes and Caffe for Scalable Deep Learning on Modern GPU Clusters," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017.

[16] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-Specialized Parameter Server," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2016.

[17] H. Zhang, Z. Zheng, W. Dai, Q. Ho, and E. P. Xing, "Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[18] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2019.

[20] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the ACM International Conference on Multimedia (MM)*, 2014.

[21] L. Bottou, "Large-Scale Machine Learning with Stochastic Gradient Descent," in *Proceedings of COMPSTAT*, 2010.

[22] A. Sergeev and M. Del Balso, "Horovod: Fast and Easy Distributed Deep Learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[23] Y. Li, M. Yu, S. Li, S. Avestimehr, N. S. Kim, and A. Schwing, "Pipe-SGD: A Decentralized Pipelined SGD Framework for Distributed Deep Net Training," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2018.

[24] A. Krizhevsky, "One Weird Trick for Parallelizing Convolutional Neural Networks," *arXiv preprint arXiv:1404.5997*, 2014.

[25] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing, "On Model Parallelization and Scheduling Strategies for Distributed Machine Learning," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2014.

[26] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: Generalized Pipeline Parallelism for DNN Training," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[27] J. H. Park, S. Kim, J. Lee, M. Jeon, and S. H. Noh, "Accelerated Training For CNN Distributed Deep Learning through Automatic Resource-Aware Layer Placement," *arXiv preprint arXiv:1901.05803*, 2019.

[28] J. H. Park, G. Yun, M. Y. Chang, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y.-r. Choi, "HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020.

[29] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zahria, and A. Talwalkar, "MLlib: Machine Learning in Apache Spark," *The Journal of Machine Learning Research*, 2016.

[30] B. Recht, C. Re, S. Wright, and F. Niu, "HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2011.

[31] X. Lian, W. Zhang, C. Zhang, and J. Liu, "Asynchronous Decentralized Parallel Stochastic Gradient Descent," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

[32] S.-Y. Zhao and W.-J. Li, "Fast Asynchronous Parallel Stochastic Gradient Descent: A Lock-Free Approach with Convergence Guarantee," in *Proceedings of the Conference on Association for the Advancement of Artificial Intelligence (AAAI)*, 2016.

[33] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2013.

[34] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware Distributed Parameter Servers," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2017.

[35] D. Harwath, A. Torralba, and J. Glass, "Unsupervised Learning of Spoken Language with Visual Context," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2016.

[36] R. Józefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, "Exploring the Limits of Language Modeling," *arXiv preprint arXiv:1602.02410*, 2016.

[37] Q. V. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng, "Building High-level Features Using Large Scale Unsupervised Learning," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2012.

[38] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[39] Y. Zhang, K. Lee, and H. Lee, "Augmenting Supervised Neural Networks with Unsupervised Objectives for Large-scale Image Classification," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2016.

[40] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *Proceedings of the Conference on International Conference on Learning Representations (ICLR)*, 2015.

[41] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons, "Proteus: Agile ML Elasticity through Tiered Reliability in Dynamic Resource Markets," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2017.

[42] Z. Jia, M. Zaharia, and A. Aiken, "Beyond Data and Model Parallelism for Deep Neural Networks," in *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2019.

[43] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2018.

[44] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device Placement Optimization with Reinforcement Learning," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2017.

[45] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," in *Proceedings of the Conference on Association for the Advancement of Artificial Intelligence (AAAI)*, 2017.

[46] Z. Xu, Y. Yang, and A. G. Hauptmann, "A Discriminative CNN Video Representation for Event Detection," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[47] S. Zha, F. Luisier, W. Andrews, N. Srivastava, and R. Salakhutdinov, "Exploiting Image-Trained CNN Architectures for Unconstrained Video Classification," *arXiv preprint arXiv:1503.04144*, 2015.

[48] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep Learning based Recommender System: A Survey and New Perspectives," *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, p. 5, 2019.

[49] J. Gu, C. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A GPU Cluster Manager for Distributed Deep Learning," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2019.

[50] M. Jeon, S. Venkataraman, J. Qian, W. Xiao, and F. Yang, "Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[51] TensorFlow, "Tensorflow benchmarks," https://github.com/tensorflow/benchmarks/.

[52] ImageNet, "ImageNet dataset," http://image-net.org/.

[53] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A Performance Model for Deep Neural Networks," in *Proceedings of the Conference on International Conference on Learning Representations (ICLR)*, 2017.

[54] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective Cluster Scheduling for Deep Learning," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[55] J. Long, E. Shelhamer, and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[56] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks," *arXiv preprint arXiv:1802.04924*, 2018.

[57] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman, "Mesh-TensorFlow: Deep Learning for Supercomputers," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2018.

[58] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-Layer CNN Accelerators," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2016.

[59] J. Bai and S. Ng, "Tests for Skewness, Kurtosis, and Normality for Time Series Data," *Journal of Business & Economic Statistics*, vol. 23, no. 1, pp. 49–60, 2005.

[60] G. Help, "Normality Testing - Skewness and Kurtosis," https://help.gooddata.com/display/doc/Normality+Testing+-+Skewness+and+Kurtosis/.

[61] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[62] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[63] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks," in *Proceedings of the Conference on International Conference on Learning Representations (ICLR)*, 2014.

[64] ImageNet, "ILSVRC2017 dataset," http://image-net.org/challenges/LSVRC/2017/.

[65] TensorFlow, "Distributed tensorflow," https://www.tensorflow.org/deploy/distributed.

[66] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[67] Bytedance Inc., "Byteps," https://github.com/bytedance/byteps.

[68] X. Wu, H. Xu, B. Li, and Y. Xiong, "Stanza: Layer Separation for Distributed Training in Deep Learning," *IEEE Transactions on Services Computing*, 2020.

[69] NVIDIA, "GeForce RTX 2060," https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2060/.

[70] ——, "Quadro P4000," https://www.nvidia.com/en-us/design-visualization/quadro-desktop-gpus/.

[71] ——, "TITAN RTX," https://www.nvidia.com/en-us/titan/titan-rtx/.

[72] ——, "TITAN V," https://www.nvidia.com/en-us/titan/titan-v/.

[73] A. V. Dorogush, V. Ershov, and A. Gulin, "CatBoost: Gradient Boosting with Categorical Features Support," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2018.

[74] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based Parameter Propagation for Distributed DNN Training," in *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2019.

[75] M. Saqib, S. D. Khan, N. Sharma, and M. Blumenstein, "A Study on Detecting Drones Using Deep Convolutional Neural Networks," in *IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, 2017.

[76] T. Wang, Y. Chen, M. Zhang, J. Chen, and H. Snoussi, "Internal Transfer Learning for Improving Performance in Human Action Recognition for Small Datasets," *IEEE Access*, 2017.

[77] F. Yu, J. Sun, A. Li, J. Cheng, C. Wan, and J. Liu, "Image Quality Classification for DR Screening Using Deep Learning," in *Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, 2017.

[78] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko, "Benchmarking and Analyzing Deep Neural Network Training," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2018.

[79] Amazon, "Amazon EC2 Pricing," https://aws.amazon.com/ec2/pricing/.

[80] Microsoft, "Microsoft Azure Pricing," https://azure.microsoft.com/en-us/pricing/.

[81] W. Jiang, G. Ye, L. T. Yang, J. Zhu, Y. Ma, X. Xie, and H. Jin, "A Novel Stochastic Gradient Descent Algorithm Based on Grouping over Heterogeneous Cluster Systems for Distributed Deep Learning," in *Proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2019.

[82] P. Patarasuk and X. Yuan, "Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations," *Journal of Parallel and Distributed Computing*, 2009.

[83] X. Chen, A. Eversole, G. Li, D. Yu, and F. Seide, "Pipelined Back-Propagation for Context-Dependent Deep Neural Networks," in *Proceedings of the Annual Conference of the International Speech Communication Association*, 2012.

[84] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing, "STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2016.

[85] J. H. Park, G. Yun, C. M. Yi, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y.-r. Choi, "HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism," *arXiv preprint arXiv:2005.14038*, 2020.

[86] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A Performance Model for Deep Neural Networks," in *Proceedings of the Conference on International Conference on Learning Representations (ICLR)*, 2017.

[87] IBM, "CPLEX-Optimizer," https://www.ibm.com/analytics/cplex-optimizer/.

[88] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[89] TensorFlow, "TensorFlow benchmarks," https://github.com/tensorflow/benchmarks/.

[90] ——, "tf.train.replica_device_setter," https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/replica_device_setter/.

[91] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization Methods for Large-Scale Machine Learning," *SIAM Review*, 2018.

[92] msr-fiddle, "Some error about communication," https://github.com/msr-fiddle/pipedream/issues/43/.

[93] ——, "PipeDream: Generalized Pipeline Parallelism for DNN Training," https://github.com/msr-fiddle/pipedream/.

[94] TensorFlow, "TFRecord and tf.train.Example," https://www.tensorflow.org/tutorials/load_data/tfrecord/.

[95] Facebook Research, "dlrm/data_utils.py," https://github.com/facebookresearch/dlrm/blob/master/data_utils.py/.

[96] Criteo Labs, "Terabyte Click Logs," https://labs.criteo.com/2013/12/download-terabyte-click-logs/.

[97] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep Learning Recommendation Model for Personalization and Recommendation Systems," *arXiv preprint arXiv:1906.00091*, 2019.

[98] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah, "Wide & Deep Learning for Recommender Systems," in *Proceedings of the workshop on deep learning for recommender systems (DLRS)*, 2016.

[99] G. Zhou, X. Zhu, C. Song, Y. Fan, H. Zhu, X. Ma, Y. Yan, J. Jin, H. Li, and K. Gai, "Deep Interest Network for Click-Through Rate Prediction," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2018.

[100] G. Zhou, N. Mou, Y. Fan, Q. Pi, W. Bian, C. Zhou, X. Zhu, and K. Gai, "Deep Interest Evolution Network for Click-Through Rate Prediction," in *Proceedings of the AAAI conference on artificial intelligence (AAAI)*, 2019.

[101] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural Collaborative Filtering," in *Proceedings of theinternational conference on World Wide Web (WWW)*, 2017.

[102] Z. Zhao, L. Hong, L. Wei, J. Chen, A. Nath, S. Andrews, A. Kumthekar, M. Sathiamoorthy, X. Yi, and E. Chi, "Recommending What Video to Watch Next: A Multitask Ranking System," in *Proceedings of the ACM Conference on Recommender Systems (RecSys)*, 2019.

[103] Y. Zhang, F. Feng, C. Wang, X. He, M. Wang, Y. Li, and Y. Zhang, "How to Retrain Recommender System? A Sequential Meta-Learning Method," in *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, 2020.

[104] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019.

[105] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang, "RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[106] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "DeepRecSys: A System for Optimizing End-To-End At-scale Neural Recommendation Inference," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[107] M. Naumov, J. Kim, D. Mudigere, S. Sridharan, X. Wang, W. Zhao, S. Yilmaz, C. Kim, H. Yuen, M. Ozdal, K. Nair, I. Gao, B.-Y. Su, J. Yang, and M. Smelyanskiy, "Deep Learning Training in Facebook Data Centers: Design of Scale-up and Scale-out Systems," *arXiv preprint arXiv:2003.09518*, 2020.

[108] R. Hwang, T. Kim, Y. Kwon, and M. Rhu, "Centaur: A Chiplet-based, Hybrid Sparse-Dense Accelerator for Personalized Recommendations," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[109] Facebook Research, "Deep Learning Recommendation Model for Personalization and Recommendation Systems," https://github.com/facebookresearch/dlrm/.

[110] GroupLens, "Movielens," https://grouplens.org/datasets/movielens/.

[111] Criteo Labs, "Kaggle Display Advertising Challenge Dataset," https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/.

[112] jayhpark530, "Parallelizing the pre-processing of the dataset." https://github.com/facebookresearch/dlrm/pull/117/.

[113] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey, "Distributed Deep Learning using Synchronous Stochastic Gradient Descent," *arXiv preprint arXiv:1602.06709*, 2016.

[114] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy, "Parameter Hub: a Rack-Scale Parameter Server for Distributed Deep Neural Network Training," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2018.

[115] A. A. Awan, C.-H. Chu, H. Subramoni, and D. K. D. Panda, "Optimized Broadcast for Deep Learning Workloads on Dense-GPU InfiniBand Clusters: MPI or NCCL?" in *Proceedings of the European MPI Users' Group Meeting(EuroMPI)*, 2018.

[116] J. Xue, Y. Miao, C. Chen, M. Wu, L. Zhang, and L. Zhou, "Fast Distributed Deep Learning over RDMA," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2019.

[117] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, T. Chen, G. Hu, S. Shaohuai, and C. Xiaowen, "Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes," *arXiv preprint arXiv:1807.11205*, 2018.

[118] M. Cho, U. Finkler, D. Kung, and H. Hunter, "BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy," in *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2019.

[119] S. Pal, E. Ebrahimi, A. Zulfiqar, Y. Fu, V. Zhang, S. Migacz, D. Nellans, and P. Gupta, "Optimizing Multi-GPU Parallelization Strategies for Deep Learning Training," *IEEE Micro*, 2019.

[120] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Exploiting Bounded Staleness to Speed Up Big Data Analytics," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2014.

[121] C.-C. Chen, C.-L. Yang, and H.-Y. Cheng, "Efficient and Robust Parallel DNN Training through Model Parallelism on Multi-GPU Platform," *arXiv preprint arXiv:1809.02839*, 2018.

[122] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour," *arXiv preprint arXiv:1706.02677*, 2017.

[123] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, L. Diao, X. Liu, and W. Lin, "DAPPLE: A Pipelined Data Parallel Approach for Training Large Models," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2021.

[124] B. Yang, J. Zhang, J. Li, C. Ré, C. Aberger, and C. De Sa, "PipeMare: Asynchronous Pipeline Parallel DNN Training," in *Proceedings of Machine Learning and Systems (MLSys)*, 2021.

[125] Q. Luo, J. Lin, Y. Zhuo, and X. Qian, "Hop: Heterogeneity-aware Decentralized Training," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[126] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2017.

[127] D. Alistarh, D. Grubic, J. Z. Li, R. Tomioka, and M. Vojnovic, "QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2017.

[128] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[129] J. Wangni, J. Wang, J. Liu, and T. Zhang, "Gradient Sparsification for Communication-Efficient Distributed Optimization," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2018.

[130] A. F. Aji and K. Heafield, "Sparse Communication for Distributed Gradient Descent," *arXiv preprint arXiv:1704.05021*, 2017.

# Acknowledgements

I would like first to thank my advisor, Professor Sam H. Noh who has continuously encouraged and supported me in all the time of research and guided me on the right track. With his mentoring and guidance, I was able to grow into an upright and good researcher. It was a great honor to be advised by Professor Sam H. Noh. Once again, I would like to express my gratitude to my esteemed advisor.

I would like to thank Professor Young-ri Choi and Professor Myeongjae Jeon who have advised the paper and dissertation committees. It was an honor and very lucky to work with them. I also would like to thank my dissertation committees, Professor Woongki Baek and Professor Jiwon Seo for their help and valuable comments.

I am grateful to my fellow labmates, Hyunsub Song, Hyeonho Song, and Eunjae Lee who have always been by my side and mean a lot to me. I am grateful to the co-authors Gyeongchan Yun, Chang M. Yi, Seungmin Lee, Nguyen T. Nguyen, Sunghwan Kim, and Jinwon Lee who worked with me.

Finally, I would like to specially thank my parents and my brother for their support and endless love.